Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Licentiate Thesis in Software and Computer Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden [2022]

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT TRITA-EECS-AVL-2022:52
ISBN 978-91-8040-327-6

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av licentiatexamen i Teknologie tisdag den 18 Oktober 2022 klockan 10:00 i D31, Lindstedtsvägen 5, Kungl Tekniska högskolan, Stockholm.

Tryck: Universitetsservice US AB

Abstract

WebAssembly has become the fourth official web language, along with HTML, CSS and JavaScript since 2019. WebAssembly allows web browsers to execute existing programs or libraries written in other languages, such as C/C++ and Rust. In addition, WebAssembly evolves to be part of edge-cloud computing platforms. Despite being designed with security as a premise, WebAssembly is not exempt from vulnerabilities. Therefore, potential vulnerabilities and flaws are included in its distribution and execution, highlighting a software monoculture problem. On the other hand, while software diversity has been shown to mitigate monoculture, no diversification approach has been proposed for WebAssembly. This work proposes software diversity as a preemptive solution to mitigate software monoculture for WebAssembly.

Besides, we provide implementations for our approaches, including a generic LLVM superdiversifier that potentially extends our ideas to other programming languages. We empirically demonstrate the impact of our approach by providing Randomization and Multivariant Execution (MVE) for WebAssembly. Our results show that our approaches can provide an automated end-to-end solution for the diversification of WebAssembly programs. The main contributions of this work are:

- We highlight the lack of diversification techniques for WebAssembly through an exhaustive literature review.
- We provide randomization and multivariant execution for WebAssembly with the implementation of two tools, CROW and MEWE respectively.
- We include constant inferring as a new code transformation to generate software diversification for WebAssembly.
- We empirically demonstrate the impact of our technique by evaluating the static and dynamic behavior of the generated diversification.

Our approaches harden observable properties commonly used to conduct attacks, such as static code analysis, execution traces, and execution time.

Keywords: WebAssembly, LLVM, Software Diversity, Automatic Software Engineering, Security

Sammanfattning

WebAssembly har sedan 2019 blivit det fjärde officiella webbspråket, tillsammans med HTML, CSS och JavaScript sedan 2019. Detta nya språk tillåter webbläsaren att köra befintliga program eller bibliotek skrivna på andra språk, såsom C/C++ och Rust. Dessutom utvecklas WebAssembly för att vara en del av edge-cloud dator -beräkningsplattformar. Trots att WebAssembly är designatd med säkerhet i fokus som en premiss är det inte undantaget från sårbarheter. Därför ingår potentiella sårbarheter och brister i dess distribution och exekvering, vilket belyser ett av problemen med mjukvarumonokultur. MÅ andra sidan, medan mångfald av programvara har visat sig mildra monokultur, har ingen diversifieringsmetod föreslagits för WebAssembly. Denna avhandling föreslår en mångfald av programvara som en förebyggande lösning med syfte att minska programvarumonokultur för WebAssembly.

Dessutom tillhandahåller vi implementeringar för våra tillvägagångssätt, däriblandinklusive en generisk LLVM- superdiversifierare som potentiellt utökar våra idéer till andra programmeringsspråk. Vi visar effekten av vårt tillvägagångssätt empiriskt genom att tillhandahålla rRandomisering och mMultivariante Exekvering (MVE) för WebAssembly. Våra resultat visar att våra tillvägagångssätt kan ge en automatiserad end-to-end lösning för diversifiering av program i WebAssembly. Detta arbetes viktigaste bidragen från detta arbete är:

- Vi lyfter fram bristen på diversifieringstekniker för WebAssembly genom en uttömmande litteraturgenomgång.
- Vi tillhandahåller en implementationeringen av två verktyg, CROW och MEWE, som genomför tillhandahåller randomisering och multivariant exekvering för WebAssembly.
- Vi inkluderar “constant inferring” som en ny kod-transformation för att generera mjukvarudiversifiering för WebAssembly.
- Vi demonstrerar empiriskt effekten av vår teknik genom att utvärdera det statiska och dynamiska beteendet hos den genererade diversifieringen.

Våra metoder härdar mot observerbara egenskaper som vanligtvis används för att utföra attacker, som statisk kodanalys, exekveringsspår och exekveringstid.

Keywords: WebAssembly, LLVM, Software Diversity, Automatic Software Engineering, Security

■ Acknowledgements

Paraphrasing a good friend of mine: the people that contributed to this work know who they are, and I prefer to thank them personally.

Javier Cabrera-Arteaga,
Stockholm, May 2022

Contents

Part I

Thesis

01 INTRODUCTION

"Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations. "

— MR. Cox ***1893***, *Cinderella: Three hundred and forty-five variants [102]*

The first web browser, Nexus [94], appeared in 1990. At that moment, web browsing was only about retrieving and showing small and static HTML web pages. In other words, users read the content of pages without interactions. The growing computing power of devices, the spread of the internet, and the need for more interaction and experiences for users encouraged the idea of executing code along with web pages. The Netscape browser made possible the execution of code on the client-side with the introduction of the JavaScript [68] language in 1995. Remarkably, all browsers have supported JavaScript since Netscape. Nowadays, most web pages include not only HTML, but also include JavaScript code that is executed in client computers. During the past decades, web browsers have become JavaScript language virtual machines. They evolved to complex systems that can run full-fledged applications, like video and audio players, animation creators, and PDF document renderers such as the one showing this document.

JavaScript is currently the most used scripting language in all modern web browsers [61]. However, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and recompiling the JavaScript code, which implies a significant overhead. Moreover, JavaScript faces security issues [75]. For example, JavaScript lacks of memory isolation, making possible to extract pieces of information from others processes [9]. Because of these problems, the Web Consortium (W3C) standardized in 2017 a bytecode for the web environment, the WebAssembly (Wasm) language.

Wasm is designed to be fast, portable, self-contained, and secure [47]. All Wasm programs are compiled ahead-of-time from source languages such as C/C++ and Rust. Wasm is created by third-party compilers that might include optimizations like in the case of LLVM. The Wasm language defines its Instruction Set Architecture [44] as an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, web browsers can use it to rapidly compile to the target architectures in a one-to-one translation process.

WebAssembly evolved outside web browsers. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [23, 39]. Consequently, in 2019, the Bytecode Alliance [34] proposed WebAssembly System Interface (WASI) [15]. WASI pioneered the execution of WebAssembly with a POSIX system interface protocol, making possible to execute Wasm directly in the operating system. Therefore, it standardizes the adoption of WebAssembly in heterogeneous platforms [29], making it suitable for edge-cloud computing platforms [6, 19]

■ 1.1 Software Monoculture

Web browsers and JavaScript have nearly three decades of development. Since then, web browsers have grown, reaching several implementations [82, 12]. Nevertheless, only Firefox, Chrome, Safari, and Edge dominate on user computers. This means that, for 5 arbitrary devices (computers, tablets, smartphones) in a world of millions, at least two of them use the same web browser. This highlights a software monoculture problem [87], as an ecosystem of machines running the exact same software. The monoculture concept is an analogy from biology [71]. It describes an ecosystem that faces extinction due the lack of diversity as all individuals share the exact same vulnerabilities. In other words, many applications can crash due to a single shared vulnerability.

Nowadays, the serving of web pages, including WebAssembly code, is centralized and provided through main servers [33]. Thus, a similar argument for software monoculture can be used for the Wasm code that is served to web browsers. Despite being designed for sandboxing and secure execution, Wasm is not exempt from vulnerabilities [27]. For example, Wasm engines are vulnerable to speculative execution [7], and C/C++ source code vulnerabilities might be ported to Wasm binaries [1]. Therefore, the sharing of the Wasm code through web browsers, also includes Wasm vulnerabilities.

The software monoculture problem escalates if we consider the edge-cloud computing platforms and how they are adopting Wasm to provide services, as we previously mentioned. Concretely, along with browser clients, thousands of edge devices running Wasm as backend services might be affected due to vulnerabilities sharing. This means that if one node in an edge network is vulnerable, all the others are vulnerable in the exact same manner as the same binary is replicated on each node. In other words, the same attacker payload would break all edge nodes at once. This illustrates how Wasm execution is fragile with respect to systemic vulnerabilities for the whole internet. Let us take the example of what happened on June 8, 2021, with Fastly [17]. That day, the whole internet suffered a 45 minutes disruption because of a failure when one Wasm binary was deployed at Fastly. The complete Fastly platform crashed. The bug, combined with most web pages being CDN-dependant, created a catastrophe. Therefore, a single distributed Wasm binary could unleash the same incident [16].

One might think that the solution is to adopt more web browser and interpreters implementations. However, this is virtually impossible as 4 web browsers dominate the market and edge-cloud computing platforms are transparently executed in the backend. Thus, a solution in this direction is doomed to fail. Another solution is to provide different WebAssembly codes. For example, a different source code, yet equivalent, can be provided when a web page requests it [14]. Consequently, millions of computers would execute different codes even though they use the same web browser. This strategy is called Software Diversification.

1.2 Software Diversification

Software Diversification is the process of finding, creating, and deploying program variants of a given original program [60] for the sake of security. Cohen et al. [93] and Forrest et al. [91] pioneered this field by proposing software diversification through code transformations. They proposed to produce variants of programs while preserving their functionalities, aiming to mitigate vulnerabilities. Since then, transformations aiming at reducing the predictability of observable behavior of programs have been proposed. For example, works on this direction proposed to diversify programs control flow [52], instruction set [89], or the system calls they use [90]. Several of these transformations can be combined to produce less predictable variants.

While previous works on software diversification demonstrated the removal of vulnerabilities, in all cases, it can be used as a preemptive solution. For example, if a vulnerability is present in one program variant, discovering and disseminating it will not affect other variants. Software diversification has been widely researched, yet, the field does not study its application to Wasm. Only Romano et al. [2] proposed the intermixing JavaScript and Wasm function calls to provide obfuscation against code analysis. For Wasm, no software diversification solution has been proposed, primarily due to its novelty.

1.3 Research questions

In this dissertation, we aim to fill the gap in the state-of-the-art of software diversification for Wasm. Three main research questions conduct our work. In this section, we present them. Our research questions are formulated by merging our publications and experiences during the creation of Software Diversification for WebAssembly.

RQ_1 **To what extent can we artificially generate program variants for WebAssembly?**
With this research question, we quantitatively assess the static differences between program variants created by our approach. We answer this question at the population level, where a program population is the collection of one original program and its generated variants. We aim to investigate the code properties that increase or diminish diversification at population level.

RQ_2 **To what extent are the generated variants dynamically different?**
With this research question, we complement RQ_1. We aim to investigate the impact on execution traces and execution times of the generated program variants.

RQ_3 **To what extent do the artificial variants exhibit different execution times on edge-cloud platforms?**
With this research question, we aim to investigate the impact of Software Diversification for WebAssembly in an emerging technology, edge-cloud computing. We evaluate the impact of a novel multivariant execution approach on real-world WebAssembly programs in a world-wide scale experiment.

1.4 Contributions

This thesis proposes four main contributions. First, as a *theoretical contribution*, we summarize the code transformations used to generate artificial software diversification through an exhaustive literature review. Consequently, we highlight the lack of diversification techniques for WebAssembly. Second, as a *technical contribution*, we provide two tools, CROW [14] and MEWE [13]. CROW creates Wasm program variants by using state-of-the-art code transformations. MEWE merges several Wasm program variants in a multivariant execution schema [60]. In addition, we summarize the main challenges faced during their implementation, such as i) program properties that make it prone to generate more variants and ii) program properties that make the observable behavior of variants different. Besides, we discuss the incorporation of a new code transformation. Third, we propose a *methodology* to quantitatively evaluate the impact of our tools, assessing the creation of artificial software diversification for WebAssembly. Fourth and final, we *empirically demonstrate* the impact of our technique by evaluating the static and dynamic behavior of the generated diversity. Our results show that creating software diversification for Wasm is feasible. Our diversification approaches affect the observable behavior such as static program properties, execution traces and execution times.

■ 1.5 Publications

This work is based on the following publications:

P_1 Superoptimization of WebAssembly Bytecode [28]
Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

P_2 CROW: Code Diversification for WebAssembly [14]
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Network and Distributed System Security Symposium (NDSS 2021), MADWeb

P_3 Multi-Variant Execution at the Edge [13]
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Under review

P_4 Scalable Comparison of JavaScript V8 Bytecode Traces [33]
Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

■ Thesis layout

This dissertation is organized in five chapters including this introduction. Chapter 2 presents the background and the state-of-the-art for WebAssembly and software diversification. Chapter 3 describes our technical contributions, the main challenges we faced and the engineering decisions carried out to implement our artifacts. Chapter 4 describes the methodology followed to answer the three main research questions driving this thesis. Chapter 5 details the main results of this work. Chapter 6 concludes and discusses future work. In addition, this dissertation contains the collection of research papers previously mentioned in this chapter.

02 BACKGROUND & STATE OF THE ART

This chapter discusses the state-of-the-art in the areas of *Wasm* and *Software Diversification*. In Section 2.1 we discuss the Wasm language, its motivation, how Wasm binaries are generated, the language specification, and security-related issues. In Section 2.2, we present a summary of Software Diversification, its foundational concepts and highlighted related works. We select the discussed works by their novelty, critical insights, and representativeness of their techniques. In Section 2.3, we finalize the chapter by highlighting open challenges in state-of-the-art related works.

■ 2.1 Wasm overview

JavaScript is currently used in all modern web browsers to allow client-side scripting. However, due to the complexity of this language, its security flaws and to gain in performance, several alternatives appeared through the years. For example, Java applets were introduced on web pages late in the 90s to execute Java bytecode in the client side [46]. Similarly, Microsoft made two attempts with ActiveX in 1996 [92], and with Silverlight in 2007 [77]. All these attempts failed to persist or had low adoption, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Alon Zakai and colleagues proposed the Emscripten tool [57]. Emscripten used a strict subset of JavaScript, asm.js, to allow low-level code such as C to be compiled to JavaScript. Asm.js was first announced as an LLVM backend [58]. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [55] compared to standard JavaScript code. Asm.js was faster than JavaScript because it limited the language features to those that can be optimized in the LLVM pipeline. Besides, it removed the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since asm.js was a subset of JavaScript it was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standardization. The work of Van Es et al. [48] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Nevertheless, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and recompiling the JavaScript code which implies a significant overhead.

Following the asm.js initiative, the W3C publicly announced the Wasm (Wasm) language in 2017. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [47] in 2017. The announcement of Wasm marked the first step into the standardization of bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [47]. Since 2017, the adoption of Wasm keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version[2]; and the case of Blazor[3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

■ 2.1.1 From source to Wasm

All Wasm programs are compiled ahead-of-time from source languages. LLVM includes Wasm as a backend since release 7.1.0 published in May 2019[4], supporting a broad range of frontend languages such as C/C++, Rust, Go or AssemblyScript[5]. The resulting binary works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In Figure 2.1, we illustrate the workflow from the creation of Wasm binaries to their execution in the browser. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations such as optimizations in the LLVM toolchain.

The step ② builds the standard library for Wasm usually as JavaScript code. This code includes the external functions that the Wasm binary needs for its execution inside the host engine. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript code. The standard library can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile both the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, since it is closer to machine code, and it is already optimized, this process is a one-to-one mapping. For instance, in the case of V8, the compilation process only applies simple and fast

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

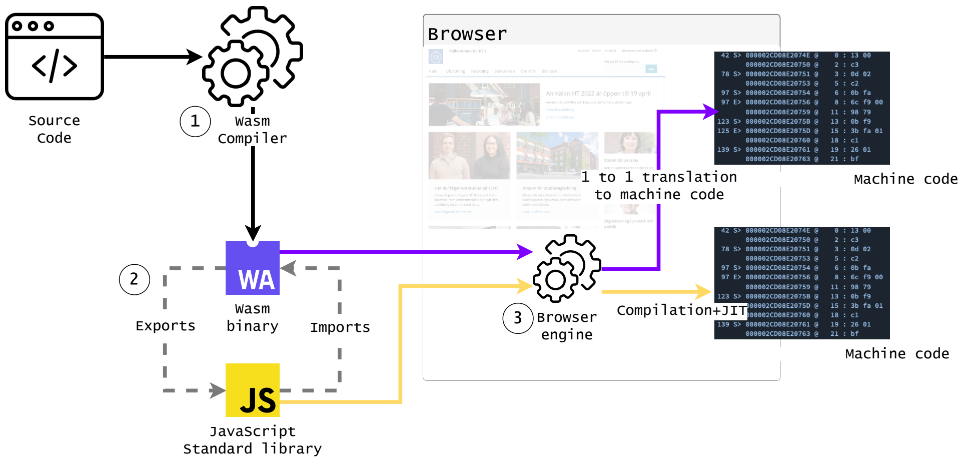
[2] `https://satoshinm.github.io/NetCraft/`

[3] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

[4] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`

[5] subset of the TypeScript language

optimizations such as constant folding and dead code removal. Once V8 completes the compilation process, the generated machine code for Wasm does not change anymore and is the same used along all its executions. This analysis was validated by conversations with the V8's development team and by experimental studies in one of our contributions [14].



```
 42 S> 000002CD08E2074E @    0 : 13 00
       000002CD08E20750 @    2 : c3
 78 S> 000002CD08E20751 @    3 : 0d 02
       000002CD08E20753 @    5 : c2
 97 S> 000002CD08E20754 @    6 : 0b fa
 97 E> 000002CD08E20756 @    8 : 6c f9 00
       000002CD08E20759 @   11 : 98 79
123 S> 000002CD08E2075B @   13 : 0b f9
125 E> 000002CD08E2075D @   15 : 3b fa 01
       000002CD08E20760 @   18 : c1
139 S> 000002CD08E20761 @   19 : 26 01
       000002CD08E20763 @   21 : bf
```

```
 42 S> 000002CD08E2074E @    0 : 13 00
       000002CD08E20750 @    2 : c3
 78 S> 000002CD08E20751 @    3 : 0d 02
       000002CD08E20753 @    5 : c2
 97 S> 000002CD08E20754 @    6 : 0b fa
 97 E> 000002CD08E20756 @    8 : 6c f9 00
       000002CD08E20759 @   11 : 98 79
123 S> 000002CD08E2075B @   13 : 0b f9
125 E> 000002CD08E2075D @   15 : 3b fa 01
       000002CD08E20760 @   18 : c1
139 S> 000002CD08E20761 @   19 : 26 01
       000002CD08E20763 @   21 : bf
```

Figure 2.1: WebAssembly is built, then compiled by the host web browser and finally executed.

Wasm can execute directly and is platform independent. Thus, the Internet of Things (IoT) can be seen as the perfect match for Wasm [7, 26] outside web browsers. IoT devices are heterogeneous in terms of architecture and platform as the same for Edge computing. For example, Singh and colleagues [31] proposed a virtual machine for Wasm in arduino based devices. On the other hand, Cloudflare and Fastly adapted their platforms to provide edge computing services directly with Wasm. In these cases, the standard library, instead of JavaScript, is provided by any other language stack that the host environment supports.

In 2019, the Bytecode Alliance [34] proposed the WebAssembly System Interface (WASI) [15]. WASI is the foundation to build Wasm code outside the browser with a POSIX system interface platform. WASI standardizes the adoption of Wasm in heterogeneous platforms [29].

■ 2.1.2 WebAssembly specification

Wasm defines its own Instruction Set Architecture (ISA) [44]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types[6], each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In Listing 2.1 and Listing 2.2 we illustrate a C program and the Wasm program that results from its compilation. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 2.2 shows the textual format for the generated Wasm. The module in this case first defines the signature of the functions (Line 2, Line 3 and Line 4) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (Line 5, Line 35 and Line 36). The definition of the function (Line 6) and its body follows the last import declaration at Line 5.

The function body is composed of local-variable declarations and typed instructions that are evaluated using a virtual stack (Line 7 to Line 32 in Listing 2.2). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at Line 32. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. As the listing shows, instructions are annotated with a numeric type.

Wasm manages the memory in a restricted way. A Wasm module has a linear memory component that is accessed with `i32` pointers (integer of 32 bits) and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in Line 37. The memory access is illustrated in Line 15. This memory is usually bound in browser engines to 4Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in Line 33 and Line 36). This ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (Line 34) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (Line 30) or using a static table of function declarations. The latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm's compiler is in charge of populating the static table of functions.

In Wasm, all instructions are grouped into blocks, where the start of a function is the root block. Two consecutive block declarations can be appreciated in Line 10 and Line 11 of Listing 2.2. Control flow structures jump between block boundaries

[6] `https://webassembly.github.io/spec/core/binary/modules.html#sections`

Listing 2.1: Example C function.

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi
                ↪ (12.54);
    }

    return A[0];
}
```

Listing 2.2: Wasm code for Listing 2.1.

```
 1 (module
 2   (type (;0;) (func (param f32) (
         ↪ result i32)))
 3   (type (;1;) (func))
 4   (type (;2;) (func (result i32)))
 5   (import "env" "ftoi" (func $ftoi (
         ↪ type 0)))
 6   (func $main (type 2) (result i32)
 7     (local i32 i32)
 8     i32.const -1000
 9     local.set 0
10     block  ;label = @1;
11       loop  ;label = @2;
12         i32.const 0
13         local.get 0
14         i32.add
15         i32.load
16         local.tee 1
17         i32.const 101
18         i32.ge_s
19         br_if 1 ;@1;
20         local.get 0
21         i32.const 4
22         i32.add
23         local.tee 0
24         br_if 0 ;@2;
25       end
26       i32.const 0
27       return
28     end
29     f32.const 0x1.9147aep+3
30     call $ftoi
31     local.get 1
32     i32.add)
33   (memory (;0;) 1)
34   (global (;4;) i32 (i32.const 1000))
35   (export "memory" (memory 0))
36   (export "A" (global 2))
37   (data $data (0) "\00\00\00\00...")
38 )
```

and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (Line 25 and Line 28). By design, each block executes independently and cannot execute or refer to outer block codes. This is guaranteed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (Line 19 and Line 24) and table break. Each break instruction can only jump to one of its enclosing blocks. For example, in Listing 2.2, Line 19 forces the execution to jump to the end of the first block that starts at Line 10 if the value at the top of the stack is greater than zero.

■ 2.1.3 WebAssembly security

As we described, Wasm is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [27] and makes it easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, implementations in both browsers and standalone runtimes [7] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [41]. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [24] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap by overflowing the stack. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for Wasm. Recently, Stiévenart and colleagues demonstrate that C/C++ source code vulnerabilities can be ported to Wasm [1]. Several proposals for extending Wasm in the current roadmap could address some existing vulnerabilities. For example, having multiple memories[7] could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors[8].

■ 2.2 Software Diversification

Software Diversification has been widely studied in the past decades. This section discusses its state-of-the-art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. According to the survey by Baudry et al. [54], the motivation for software diversification can be separated in five categories: reusability [81], software testing [69], performance [65], fault tolerance [97] and security [93]. Our work contributes to the latter two categories. In this section we discuss related works by highlighting how they generate diversification and how they put it into practice.

There are two primary sources of software diversification: Natural Diversity and Artificial Diversity[54]. This work contributes to the state of the art of Artificial Diversity, which consists of software synthesis. This thesis is founded on the work of Cohen in 1993 [93] as follows.

[7]`https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md`

[8]`https://webassembly.org/roadmap/`

■ 2.2.1 Variants' generation

Cohen et al. [93] proposed to generate artificial software diversification through mutation strategies. A mutation strategy is a set of rules to define how a specific component of software development should be changed to provide a different yet functionally equivalent program. Cohen and colleagues proposed 10 concrete transformation strategies that can be applied at fine-grained levels. All described strategies can be mixed together. They can be applied in any sequence and recursively, providing a richer diversity environment. We summarize them, complemented with the work of Baudry et al. [54] and the work of Jackson et al. [63], in 5 strategies.

(S1) *Equivalent instructions replacement* Semantically equivalent code can replace pieces of programs. This strategy replaces the original code with equivalent arithmetic expressions or injects instructions that do not affect the computation result. There are two main approaches for generating equivalent code: rewriting rules and exhaustive searching. The replacement strategies are written by hand as rewriting rules for the first one. A rewriting rule is a tuple composed of a piece of code and a semantic equivalent replacement. For example, Cleemput et al. [64] and Homescu et al. [62] insert NOP instructions to generate statically different variants. In their works, the rewriting rule is defined as `instr => (nop instr)`, meaning that `nop` operation followed by the instruction is a valid replacement . On the other hand, exhaustive searching samples all possible programs for a specific language. In this topic, Jacob et al. [73] proposed the technique called superdiversification for x86 binaries. The superdiversification strategy proposed by Jacob and colleagues performs an exhaustive search of all programs that can be constructed from a specific language grammar. If one of the generated programs is equivalent to the original program, then it is reported as a variant. Similarly, Tsoupidi et al. [20] introduced Diversity by Construction, a constraint-based compiler to generate software diversity for MIPS32 architecture.

(S2) *Instruction reordering* This strategy reorders instructions or entire program blocks if they are independent. The location of variable declarations might change as well if compilers re-order them in the symbol tables. It prevents static examination and analysis of parameters and alters memory locations. In this field, Bhatkar et al. [88, 83] proposed the random permutation of the order of variables and routines for ELF binaries.

(S3) *Adding, changing, removing jumps and calls* This strategy creates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [93] mainly illustrated the case by inserting bogus jumps in programs. Pettis and Hansen [95] proposed to split basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [53] de-inline basic blocks of code as an LLVM pass. In their approach, each de-inlined code is transformed into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [83] extended their previous approach [88], replacing function calls by indirect pointer calls in C source code, allowing post binary reordering of function calls. Recently, Romano et al. [2] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm function.

(S4) *Program memory and stack randomization* This strategy changes the layout of programs in the host memory. Also, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [88, 83] propose to randomize the base addresses of applications and the library memory regions in ELF binaries. Tadesse Aga and Autin [35], and Lee et al. [8] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [79] propose to separate a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [18] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization.

(S5) *ISA randomization and simulation* This strategy uses a key to cypher the original program binary into another encoded binary. Once encoded, the program can be decoded only once at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [86], and Barrantes et al. [89] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [90] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [50] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Code obfuscation [2] can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as is in any client. Yet, both strategies are meant to tackle program analysis from potential attackers.

■ 2.2.2 Variants' equivalence

Equivalence checking between program variants is an essential component for any program transformation task, from checking compiler optimizations [59] to the artificial synthesis of programs discussed in this chapter. Equivalence checking proves that two pieces of code or programs are semantically equivalent [32]. Cohen [93] simplifies this checking by enunciating the following property: two programs are equivalent if given identical input, they produce the identical output. We use this same enunciation as the definition of *functional equivalence* along with this dissertation. Equivalence checking in Software Diversification aims to preserve the original functionality for programs while changing observable behaviors. For example, two programs can be statically different or have different execution times and provide the same computation.

The equivalence property is often guaranteed by construction. For example, in the case illustrated in S1 for Cleemput et al. [64] and Homescu et al. [62], their transformation strategies are designed to generate semantically equivalent program variants. However, this process is prone to developer errors, and further validation is needed. For example, the test suite of the original program can be used to check the variant. If the test suite passes for the program variant [25], then this variant can be considered equivalent to the original. However, this technique is limited due to the need for a preexisting test suite. When the test suite does not exist, another technique is needed to check for equivalence.

If there is no test suite or the technique does not inherently implement the equivalence property, the previously mentioned works use theorem solvers (SMT solvers) [74] to prove equivalence. For SMT solvers, the main idea is to turn the two code variants into mathematical formulas. The SMT solver checks for counter-examples. When the SMT solver finds a counter-example, there exists an input for which the two mathematical formulas return a different output. The main limitation of this technique is that all algorithms cannot be translated to a mathematical formula, for example, loops. Yet, this technique tends to be the most used for no-branching-programs checking like basic block and peephole replacements [49].

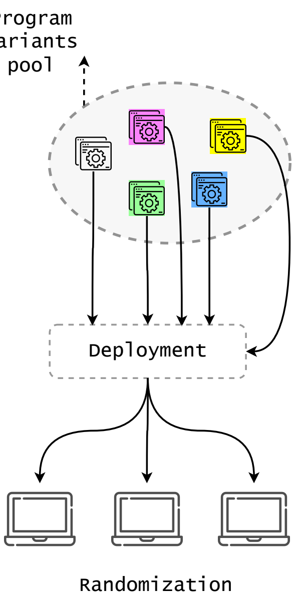
Another approach to check equivalence between two programs similar to using SMT solvers is by using fuzzers [43]. Fuzzers randomly generate inputs that provide different observable behavior. If two inputs provide a different output in the variant, the variant and the original program are not equivalent. The main limitation for fuzzers is that the process is remarkably time-expensive and requires the introduction of oracles by hand.

■ 2.2.3 Usages of Software Diversity

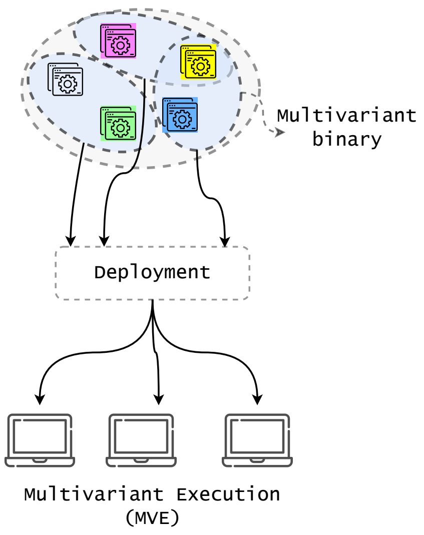
After program variants are generated, they can be used in two main scenarios: Randomization or Multivariant Execution (MVE) [63]. In Figure 2.2a and Figure 2.2b we illustrate both scenarios.

(U1) *Randomization:* In the context of our work *Randomization* refers to the ability of a program to be served as different variants to different clients. In the scenario of Figure 2.2a, a program is selected from the collection of variants (program's variant pool), and at each deployment, it is assigned to a random client. Jackson et al. [63] compare the variant's pool in Randomization with a herd immunity, since vulnerable binaries can affect only part of the client's community.

El-Khalil and colleagues [84] propose to use a custom compiler to generate different binaries out of the compilation process. El-Khalil and colleagues modify a version of GCC 4.1 to separate a conventional stack into several component parts,



(a) Randomization scenario. Given a pool of program variants, one variant is deployed per host. Each deployment randomly selects which variant is assigned to each host. The same program variant is executed in the host at every program invocation between deployments.



(b) Multivariant Execution scenario. Given a pool of program variants, a sample of the pool is packaged in a multivariant binary that is deployed. Each deployment randomly selects which multivariant binary is assigned to each host. Finally, a variant from the multivariant binary is randomly executed at runtime in the host.

Figure 2.2: Software Diversification usages.

called multistacks. On the same topic, Aga and colleagues [35] propose to generate program variants by randomizing its data layout in memory. Their approach makes each variant to operate the same data in memory with different memory offsets. The Polyverse company[9] materializes randomization at the commercial level in real life. They deliver a unique Linux distribution compilation for each of its clients by scrambling the Linux packages at the source code level.

Virtual machines and operating systems can be also randomized. On this topic, Kc et al. [86], create a unique mapping between artificial CPU instructions and real ones. Their approach makes possible the assignment of different variants to specific target clients. Similarly, Xu et al. [18] recompile the Linux Kernel to reduce the

[9] `https://polyverse.com/`

exposure time of persistent memory objects, increasing the frequency of address randomization.

(U2) *Multivariant Execution (MVE):* Multiple program variants are composed in one single binary (multivariant binary) [80]. Each multivariant binary is randomly deployed to a client. Once in the client, the multivariant binary executes its embedded program variants at runtime. Figure 2.2b illustrates this scenario.

The execution of the embedded variants can be either in parallel to check for inconsistencies or a single program to randomize execution paths [88]. Bruschi et al. [78] extended the idea of executing two variants in parallel with not-overlapping and randomized memory layouts. Simultaneously, Salamat et al. [76] modified a standard library that generates 32-bit Intel variants where the stack grows in the opposite direction, checking for memory inconsistencies. Notably, Davi et al. proposed Isomeron [52], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. The previously mentioned works showed the benefits of exploiting the limit case of executing only two variants in a multivariant environment. Agosta et al. [56] and Crane et al. [53] used more than two generated programs in the multivariant composition, randomizing software control flow at runtime.

Both scenarios have demonstrated to harden security by tackling known vulnerabilities such as (JIT)ROP attacks [66] and power side-channels [67]. Moreover, Artificial Software Diversification is a preemptive technique for yet unknown vulnerabilities [63]. Our work contributes to both usage scenarios for Wasm.

■ 2.3 Open challenges

In Table 2.1 we list the related work on Artificial Software Diversification discussed along with this chapter. The first column in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. In the following text, we enumerate the open challenges we have found in the literature research:

1. *Software monoculture:* The same Wasm code is executed in millions of clients devices through web browser. In addition, Wasm evolves to support edge-cloud computing platforms in backend scenarios, *i.e.,* replicating the same binary along with all computing nodes in a worldwide scale. Therefore,

potential vulnerabilities are spread, highlighting a monoculture phenomenon [3].

2. *Lack of Software Diversification for Wasm*: Software Diversification has demonstrated to provide protection for known and yet-unknown vulnerabilities. However, only one software diversity approach has been applied to the context of Wasm [2]. Moreover, Wasm is a novel technology and, the adoption of defenses is still under development [7, 10] and has a low pace, making software diversification a possible preemptive technique. Besides, the preexisting works based on the LLVM pipeline cannot be extended to Wasm because they contribute to LLVM versions released before the inclusion of Wasm as an architecture.

3. *Lack of research on MVE for Wasm:* Wasm has a growing adoption for Edge platforms. However, MVE in a distributed setting like the Edge has been less researched. Only Voulimeneas et al. [5] recently proposed a multivariant execution system by parallelizing the execution of the variants in different machines for the sake of efficiency.

■ Conclusions

In this chapter, we presented the background on the Wasm language, including its security issues and related work. This chapter aims to settle down the foundation to study automatic diversification for Wasm. We highlighted that Wasm has been less researched in the field of Artificial Software Diversification. On the other hand, current available implementations for Software Diversification cannot be directly ported to Wasm. The current limitations on security and the lack of software diversity approaches for Wasm motivate our work. We place our contributions in the field of artificial diversity. In Chapter 3 we describe the technical details that lead our contributions. Besides, the impact of our contributions is evaluated by following the methodology described in Chapter 4.

Authors	**S1**	**S2**	**S3**	**S4**	**S5**	**U1**	**U2**	**Main technical contribution**
Pettis and Hansen [95]		✓		✓		✓		Custom Pascal compiler for PA-RISC architecture
Chew and Song [90]			✓			✓		Linux Kernel recompilation.
Kc et al. [86]					✓			Linux Kernel recompilation.
Barrantes et al. [89]					✓	✓		x86 to x86 transformations using Valgrind
Bhatkar et al. [88]	✓	✓		✓		✓		ELF binary transformations
El-Khalil and Keromytis [84]						✓		custom GCC compiler for x86 architecture
Bhatkar et al. [83]	✓	✓		✓		✓		C/C++ source to source transformations and ELF binary transformations
Younan et al. [79]				✓				custom GCC compiler
Bruschi et al. [78]				✓		✓		ELF binary transformations.
Salamat et al. [76]			✓				✓	Custom GNU compiler
Jacob et al. [73]	✓	✓						x86 to x86 transformations
Salamat et al. [70]				✓			✓	x86 to x86 transformations
Amarilli et al. [67]	✓				✓	✓		Polymorphic code generator for ARM architecture
Jackson [63]	✓					✓	✓	LLVM compiler, only backend for x86 architecture
Cleemput et al. [84]	✓					✓		x86 to x86 transformations
Homescu et al. [62]	✓					✓		LLVM 3.1.0†
Crane et al. [53]	✓	✓	✓				✓	LLVM, only backend for x86 architecture
Davi et al. [52]							✓	Windows DLL instrumentation
Couroussé et al. [50]	✓	✓			✓	✓		Custom GCC compiler targeting microcontrollers
Lu et al. [37]				✓			✓	GNU assembler for Linux kernel
Belleville et al. [42]	✓			✓		✓		Only C language frontend, LLVM 3.8.0†
Aga et al. [35]				✓		✓		Data layout randomization, LLVM 3.9†
Österlund et al. [30]				✓			✓	Linux Kernel recompilation.
Xu et al. [18]				✓		✓		Custom kernel module in Linux OS
Lee et al. [8]				✓		✓		LLVM 12.0.0 backend for x86
Romano et al. [2]			✓			✓		JavaScript and Wasm intermixing

† Notice that LLVM only supports Wasm backend from release 7.1.0

Table 2.1: The first columns in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order.

03 AUTOMATIC DIVERSITY FOR WASM

We aim to create artificial software diversity for Wasm by providing methods and tools to make the process easier and feasible for developers and researchers. According to our exhaustive literature review, no paper provides artificial software diversification for WebAssembly. Therefore, we need to enunciate the engineering foundation to implement the strategies defined in Section 2.2. Our implementations are part of the contributions of this thesis. We provide two tools that complement this work: CROW and MEWE. The former tool generates Wasm program variants statically at compile time to provide randomization. The latter tool provides the method to generate MVE binaries for WebAssembly. In this chapter, we describe our technical contributions. In Section 3.1 we enunciate how the current state-of-the-art leads us to contribute with Software Diversification through LLVM. We follow by describing our tools and their main technical insights in Section 3.2 and Section 3.3. In addition, we describe a new transformation strategy as part of our contributions.

■ 3.1 Global approach

The work of Hilbig et al. [11] in 2021 influences our design decisions. According to their work, 70% of the Wasm binaries in the wild are created with LLVM-based compilers. Therefore, we provide artificial software diversity for Wasm through LLVM. Other solutions would have been to diversify at the source-code level or the Wasm binary level. However, these facts would limit the applicability of our work. Our approach is more general as diversification also will work for other LLVM backends.

LLVM is a compound of three main components [85]. First, the frontend (compilers such as clang and rustc) converts the program source code to LLVM intermediate representation (LLVM IR). Second, optimization and transformation processes improve the LLVM IR. Third and final, the backend component is in charge of generating the target machine code. In Figure 3.1 we show how we use the LLVM pipeline in our contributions, which are highlighted as dashed squares.

The global workflow in Figure 3.1 starts by receiving the source code. Then the LLVM frontend transforms it into LLVM IR representation ①. We alter the

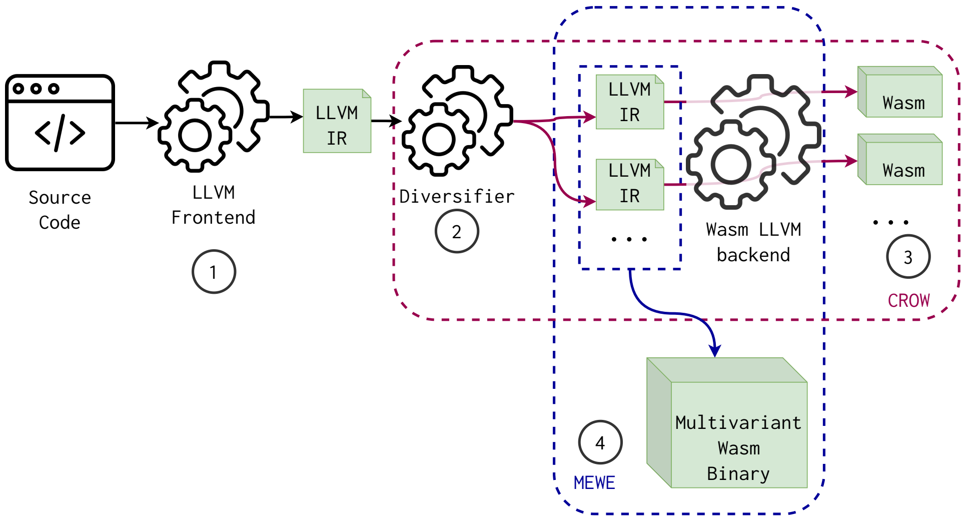


Figure 3.1: Generic workflow to create Wasm program variants.

LLVM pipeline that compiles source code to Wasm by introducing a diversifier component.

The diversifier generates LLVM IR variants from the output of the frontend (2). The LLVM IR variants are inputs for our customized Wasm backend. The diversifier and the custom Wasm LLVM backend compose CROW, which creates Wasm program variants out of a source code program (3). In addition, an orthogonal tool comes from the generation of LLVM IR variants at Step (2). MEWE [13], merges and creates multivariant binaries to provide MVE for Wasm (4).

3.2 CROW: Code Randomization of WebAssembly

This section describes the red squared tooling in Figure 3.1 named CROW [14]. CROW is a tool tailored to create semantically equivalent Wasm variants from an LLVM front-end output. Using a custom Wasm LLVM backend, it generates the Wasm binary variants.

In Figure 3.2, we describe the workflow of CROW to create program variants. The Diversifier in CROW is composed by two main processes, *exploration* and *combining*. The *exploration* process operates at the instruction level for each function in its input LLVM. For all LLVM instructions, CROW produces a collection of functionally equivalent code replacements. In the *combining* stage, CROW assembles the code replacements to generate different LLVM IR variants. CROW generates the LLVM IR variants by traversing the power set of all possible combinations of code replacements. Finally, the custom Wasm LLVM backend compiles the assembled LLVM IR variants into Wasm binaries. In the following text, we describe our design decisions. All our implementation choices are based on one premise: *each design decision should increase the number of Wasm variants that CROW creates.*

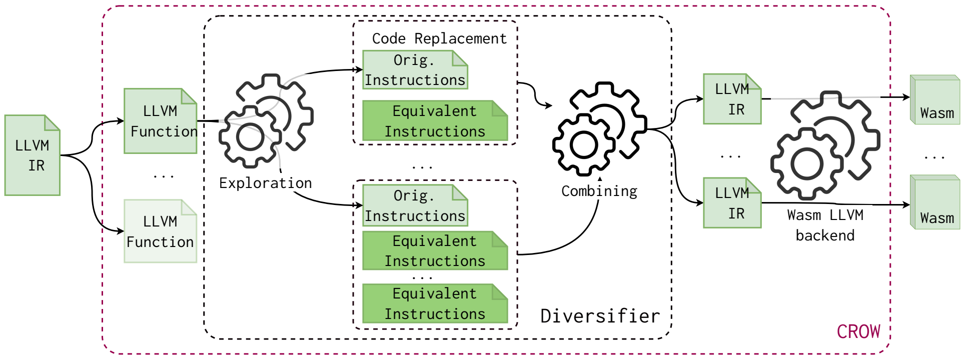


Figure 3.2: CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

■ 3.2.1 Exploration

The primary component of CROW's exploration process is its code replacements generation strategy. The diversifier implemented in CROW is based on the proposed superdiversifier of Jacob et al. [73]. A superoptimizer focuses on *searching* for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [96] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by code size in ascending order. If any of these programs is found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We use the superdiversifier idea of Jacob and colleagues to implement CROW because of two main reasons. First, the code replacements generated by this technique outperform diversification strategies based on handwritten rules. Concretely, we can control the quality of the generated codes. Besides, CROW always generates equivalent programs because it is based on a solver to check for equivalence. Second, there is a battle-tested superoptimizer for LLVM, Souper [45]. This latter makes it feasible the construction of a generic LLVM superdiversifier.

We modify Souper to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation is the optimal replacement result of the searching. CROW keeps more equivalent replacements during the searching by removing the halting criteria. Instead the original halting conditions, CROW does not halt when it finds the first replacement. CROW continues the search until a timeout is reached or the replacements grow to a size larger that a predefined threshold.

Notice that the searching space increases exponentially with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions with no correspondence in the Wasm backend. This decision reduces the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more program variants. For example, Souper prevents the generation of the commutative operations during the searching. On the contrary, CROW still uses such transformation as a strategy to generate program variants.

■ 3.2.2 Constant inferring

One of the code transformation strategies of Souper does *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on variables that are used to control branches. By extending Souper as a superdiversifier, we add this transformation strategy as a new mutation strategy to the ones defined in Section 2.2.

After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification. Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```
int babbage() {
    int current = 0,
        square;
    while ((square=current*current) %
            ↪ 1000000 != 269696) {
        current++;
    }
    printf ("The number is %d\n", current)
            ↪ ;
    return 0 ;
}
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1.

```
int babbage() {
    int current = 25264;



    printf ("The number is %d\n", current);
    return 0 ;
}
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large. The original Souper deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[1].

■ 3.2.3 Removing subsequent optimizations for LLVM

During the implementation of CROW, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. Therefore, we modify the Wasm backend. We disable all optimizations in the Wasm backend that could reverse the CROW transformations. In the following enumeration, we list three concrete optimizations that we remove from the Wasm backend.[2]

- Constant folding: this optimization calculates the operation over two (or more) constants in compiling time, and replaces the original expression by its constant result. For example, let us suppose $a = 10 + 12$ a subexpression to be compiled, with the original optimization, the Wasm backend replaces it by $a = 22$.
- Expressions normalization: in this case, the comparison operations are normalized to its complementary operation, e.g. $a > b$ is always replaced by $b <= a$.
- Redundant operation removal: expressions such as the multiplication of variables by $a = b2^n$ are replaced by shift left operations $a = b << n$.

[1]Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also, notice that the two programs in the example follow the definition of *functional equivalence* discussed in Section 2.2.

[2]We only illustrate three of the removed optimization for the sake of simplicity.

■ 3.3 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [13]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE). The tool generates application-level multivariant binaries without changing the operating system or Wasm runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as step ② in Figure 3.1 shows. CROW generates each one of these variants with fine-grained diversification at the instruction level, applying the majority of the strategies discussed in Section 2.2 and *constant inferring*. MEWE adds a new mutation strategy. It inlines function variants when appropriate, resulting in call stack diversification at runtime.

In Figure 3.3 we zoom MEWE from the blue highlighted square in Figure 3.1. MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW and creates an LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, allowing the execution path randomization.

The Mixer augments the LLVM multivariant binary with a random generator. The random generator is needed to perform the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, MEWE generates a standalone multivariant Wasm binary using the same custom Wasm LLVM backend from CROW. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

■ 3.3.1 Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* N *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* f_2 *by* f_1 *[98]. The nodes in* N *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom

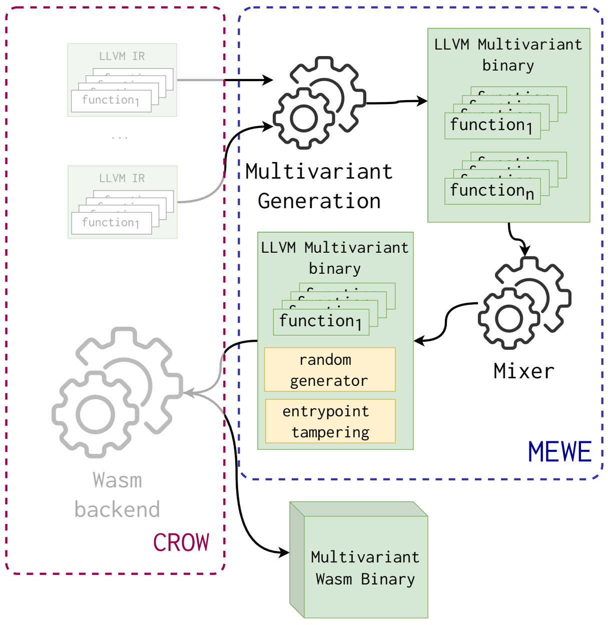


Figure 3.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed.

of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

In Listing 3.3, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. It first calls the random generator, which returns a value used to invoke a specific function variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution-based attacks [7]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [10].

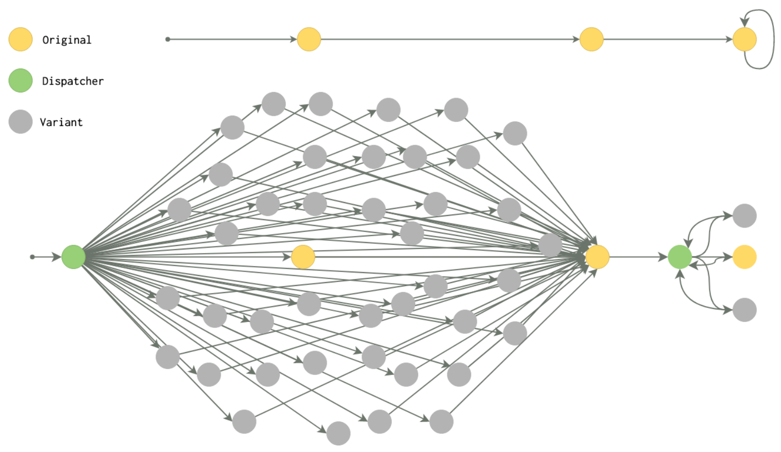


Figure 3.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow).

This also allows MEWE to inline function variants inside the dispatcher instead of defining them again. Here we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.3: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

■ 3.3.2 The Mixer

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the application's entrypoint, and merge all these components into a multivariant Wasm binary. We use the Rustc compiler[3] to orchestrate the mixing. For the random generator, we rely on WASI's specification [15] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. The Mixer creates a new entrypoint for the binary called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

■ 3.4 Accompanying Source Code

This thesis is accompanied by the source code of both contributions, CROW and MEWE. The source code is accessible through the links:

1. CROW: `https://github.com/KTH/slumps`
2. Customized LLVM backend: `https://github.com/Jacarte/llvm-project`
3. MEWE: `https://github.com/Jacarte/MEWE`

Our software artifacts are licensed under the MIT License. The dependent source codes, such as LLVM, are licensed under their original conditions.

■ Conclusions

This chapter discusses the technical details of the tools implemented for our main contributions. We describe how CROW generates program variants for the sake of software diversification. We propose a global architecture for a generic LLVM superdiversifier We introduce a new mutation strategy that is a consequence of retargeting Souper as a superdiversifier. Besides, we dissect MEWE and how it creates an MVE system. In Chapter 4 we discuss the methodology we follow to evaluate how CROW and MEWE create software diversification.

[3]`https://doc.rust-lang.org/rustc/what-is-rustc.html`

04 METHODOLOGY

In this chapter, we present our methodology to answer the research questions enunciated in Section 1.3. We investigate three research questions. In the first question, we aim to investigate the static differences between variants. We evaluate the code properties that increase or diminish software diversification. Our second research question focuses on comparing their behavior during their execution, complementing our first research question. The generated variants should be statically different, but also should provide different observable behaviors. The final research question evaluates the feasibility of using the program variants in security-sensitive environments. We evaluate our generated program variants in an edge-cloud computing platform proposing a novel multivariant execution approach.

The main objective of this thesis is to study the feasibility of automatically creating program variants out of preexisting program sources. To achieve this objective, we use the empirical method by Runeson et al. [22], using the prototype solutions discussed in Chapter 3 and evaluating them through quantitative analyses in case studies. We follow an iterative and incremental approach on the selection of programs for our corpora. To build our corpora, we find a representative and diverse set of programs to generalize, even when it is unrealistic following an empirical approach, as much as possible our results. We first enunciate the corpora we share along this work to answer our research questions. Then, we establish the metrics for each research question, set the configuration for the experiments, and describe the protocol.

■ 4.1 Corpora

Our experiments assess the impact of artificially created diversity. The first step is to build a suitable corpus of programs' seeds to generate the variants. Then, we answer all our research questions with three corpora which follow two main properties: 1) *functionally diverse:* the selection of the programs is not biased by functionally fixed tasks, for example, the Rosetta contains programs that solve from the *Babbage* problem to calculate *Convex Hull*; and 2) *representative:* our corpora have 3021 programs that can be ported to Wasm, representing approximately 40% of the unique Wasm binaries in the wild [11].

We build our three corpora in an escalating strategy based on the merging of our previous publications. The first corpus is diverse and contains simple programs in terms of code size, making them easy to manually analyze. The second corpus is a project meant for security-sensitive applications. The third corpus is a QR encoding decoding algorithm. In the following text, we describe the filtering and description of each corpus.

1. **Rosetta**: We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains many tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from the Rosetta Code, representing 989 programs as of 01/26/2020. We then apply several filters: the programs should successfully compile and, they should not require user inputs for automatic execution, the programs should terminate and should not result in non-deterministic results.

   The result of the filtering is a corpus of 303 C programs. All programs include a single function in terms of source code. These programs range from 7 to 150 lines of code.

2. **Libsodium**: This project is an encryption, decryption, signature, and password hashing library implemented in 102 separated modules. The modules have between 8 and 2703 lines of code per function. This project is selected based on two main criteria: first, its importance for security-related applications, and second, its suitability to collect the modules in LLVM intermediate representation.

3. **QrCode**: This project is a QrCode and MicroQrCode generator written in Rust. This project contains 2 modules having between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for collecting the modules in their LLVM representation. This project increases the complexity of the previously selected projects due to its integration with image generation.

In Table 4.1 we listed the corpus name, the language of the programs in the corpus, the number of modules, the total number of functions, the range of lines of code, and the original location of the corpus.

■ 4.2 RQ_1. To what extent can we artificially generate program variants for WebAssembly?

This research question investigates whether we can artificially generate program variants for Wasm. We use CROW to generate variants from an original program, written in C/C++ in the case of the Rosetta corpus and LLVM bitcode modules in the case of the Libsodium and QrCode. In Figure 4.1 we illustrate the workflow to generate Wasm program variants. We pass each function of the corpora to CROW

[1]`http://www.rosettacode.org/wiki/Rosetta_Code`

Corpus	Lang.	No. modules	No. functions	LOC range	Location
Rosetta	C	-[2]	303	7 - 150	`https://github.com/KTH/slumps/tree/master/benchmark_programs/rossetta/valid/no_input`
Libsodium	LLVM IR + Rust	102	869	8 - 2703	`https://github.com/jedisct1/libsodium/tree/2b5f8f2b6810121c2d9a8cc8a392e01f4d3de433`
QrCode	LLVM IR + Rust	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust/commit/faa4397ba7c5f441cb9a2b436c1e84a0d52ae942`
Total			3021		

Table 4.1: Corpora description. The table is composed by the name of the corpus, programming language of the programs in the corpus, the number of modules, the number of functions, the lines of code range and the location of the corpus.

as a program to diversify. To answer RQ1, we study the outcome of this pipeline, the generated Wasm variants.

■ Metrics

To assess our approach's ability to generate Wasm binaries that are statically different, we compute the number of variants and the number of unique variants for each original function of each corpus. On top, we define the aggregation of these former two values to quantitatively evaluate RQ1 at the corpus level.

We start by defining what a program's population is. This definition can be applied in general to any collection of variants of the same program. All definitions are based upon bytecodes and not the source code of the programs.

Definition 2. *Program's population $M(P)$:* Given a program P and its generated variants v_i, the program's population is defined as:

$$M(P) = \{v_i \text{ where } v_i \text{ is a variant of P}\}$$

Notice that the program's population includes the original program P.

Beyond the program's population, we also want to compare how many program variants are unique. The subset of unique programs in the program's population hints how the variants are different between them and not only against the original

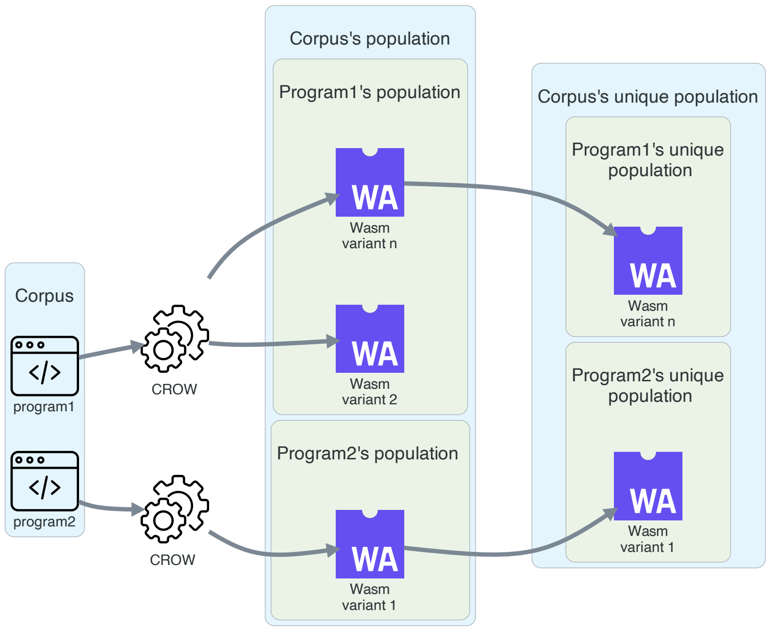


Figure 4.1: The program variants generation for RQ1.

program. For example, imagine a program P with two program variants V_1 and V_2, the program population is composed by $\{P, V_1$ and $V_2\}$, where V_1 is different from P, and V_2 is different from P. V_1 is either equal or different from V_2, the program's population still be the same. If V_1 and V_2 are equal, then only one unique variant is generated,

Definition 3. *Program's unique population* $U(P)$*:* Given a program P and its program's population $M(P)$, the program's unique population is defined as.

$$U(P) = \{v \ \in \ M(P)\}$$

such that $\forall v_i, v_j \in U(P)$, $v_i \neq v_j \Rightarrow md5sum(v_i) \neq md5sum(v_j)$. $Md5sum(v)$ is the md5 hash calculated over the bytecode stream of the program file v. Notice that the original program P is included in $U(P)$.

Metric 1. *Program's population size* $S(P)$*:* Given a program P and its program's population $M(P)$ according to Definition 2, the program's population size is defined as.

$$S(P) = |M(P)|$$

Metric 2. *Program's unique population size $US(P)$:* Given a program P and its program's unique population $U(P)$ according to Definition 3, the program's unique population size is defined as.

$$US(P) = |U(P)|$$

Metric 3. *Corpus population size $CS(C)$:* Given a program's corpus C, the corpus population size is defined as the sum of all program's population sizes over the corpus C:

$$CS(C) = \Sigma S(P) \; \forall \; P \; \in \; C$$

Metric 4. *Corpus unique population size $UCS(C)$:* Given a program's corpus C, the corpus unique population size is defined as the sum of all program's unique population sizes over the corpus C :

$$UCS(C) = \Sigma US(P) \; \forall \; P \; \in \; C$$

■ Protocol

To generate program variants, we synthesize programs with an enumerative strategy, checking each synthesis for equivalence modulo input [38] against the original program, as it is described in Section 3.2. For obvious reasons, this space is nearly impossible to explore in a reasonable time as soon as the limit of instructions increases. Therefore, we use two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all instructions in the CROW diversifier synthesis.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus and the properties of its programs, we set the exploration time to 1 hour maximum per function for Rosetta. In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function. The decision behind the usage of lower timeout for Libsodium and QrCode is motivated by the properties listed in Table 4.1. The latter two corpora are remarkably larger regarding the number of instructions and functions.

We pass each of the 303 + 869 + 1849 functions in the corpora to CROW, as Figure 4.1 illustrates, to synthesize program variants. We calculate the *Corpus population size* (Metric 3) and *Corpus unique population size* (Metric 4) for each corpus and conclude by answering RQ1.

■ 4.3 RQ_2. To what extent are the generated variants dynamically different?

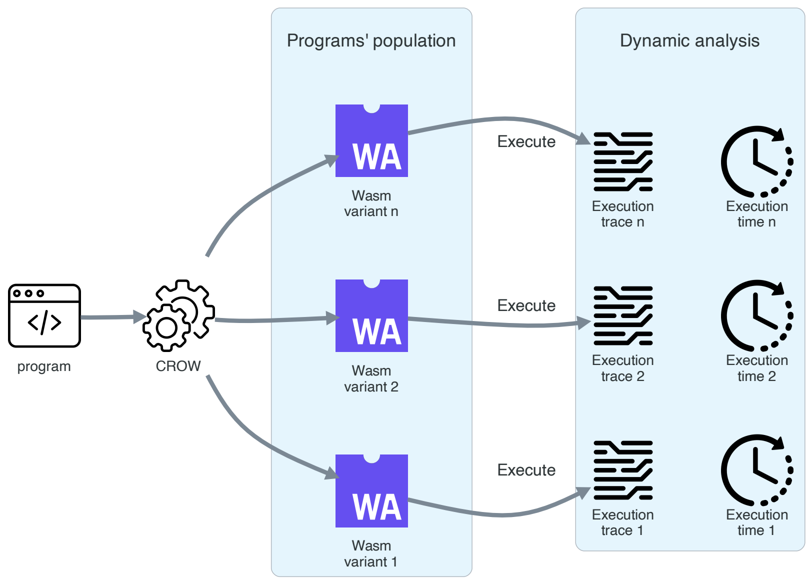


Figure 4.2: Dynamic analysis for RQ2.

In this second research question, we investigate to what extent the artificially created variants are dynamically different between them and in comparison to the original program. To conduct this research question, we could separate our experiments into two fields as Figure 4.2 illustrates: static analysis and dynamic analysis. The static analysis focuses on the appreciated differences among the program variants, as well as between the variants and the original program. We perform the static analysis in answering RQ1 in Section 4.2. With RQ2, we focus on the last category, the dynamic analysis of the generated variants. This decision is supported because dynamic analysis complements RQ1 and, it is essential to provide a full understanding of diversification. We use the original functions from the Rosetta corpus described in Section 4.1 and their variants generated to answer RQ1. We use only Rosetta to answer RQ2 because this corpus is composed of simple programs that can be executed directly without user interaction, *i.e.,* we only need to call the interpreter passing the Wasm binary to it. To dynamically compare programs and their variants, we execute each program on each programs' population to collect and execution times. We define execution trace and execution time in the following section.

■ Metrics

We compare the execution traces of two any programs of the same population with a global alignment metric. We propose a global alignment approach using Dynamic Time Warping (DTW). Dynamic Time Warping [99] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. DTW has been used for comparing traces in different domains. For software, De A. Maia et al. [72] proposed to identify similarity between programs from execution traces. As we discussed in Section 2.1, a theoretical Wasm engine perform `push` and `pop` operations when the program instructions are executed. Therefore, in our experiments, we define the execution traces as the sequence of the stack operations during the execution of the Wasm program. In the following text, we define the $TraceDiff$ metric.

Metric 5. *TraceDiff :* Given two programs P and P' from the same program's population, TraceDiff (P,P'), computes the DTW distance collected during their execution.
A TraceDiff of 0 means that both traces are identical. The higher the value, the more different the traces.

Moreover, we use the execution-time distribution of the programs in the population to complement the answer to RQ2. For each program pair in the programs' population, we compare their execution-time distributions. We define the execution time as follows:

Metric 6. *Execution time:* Given a Wasm program P, the execution time is the time spent to execute the binary.

■ Protocol

To compare program and variants behavior during runtime, we analyze all the unique program variants generated to answer RQ1 in a pairwise comparison using the value of aligning their execution traces (Metric 5). We use SWAM[3] to execute each program and variant to collect the stack operation traces. SWAM is a Wasm interpreter that provides functionalities to capture the dynamic information of Wasm program executions, including the virtual stack operations.

Furthermore, we collect the execution time, Metric 6, for all programs and their variants. We compare the collected execution-time distributions between programs using a Mann-Withney U test [101] in a pairwise strategy.

[3]`https://github.com/satabin/swam`

■ 4.4 RQ_3. To what extent do the artificial variants exhibit different execution times on edge-cloud platforms?

To answer RQ3, we use the variants generated for the programs of Libsodium and QrCode corpora, we take 2+5 programs interconnecting the LLVM bitcode modules (mentioned in Table 4.1). We illustrate the protocol to answer RQ3 in Figure 4.3 starting from the creation of the programs' population.

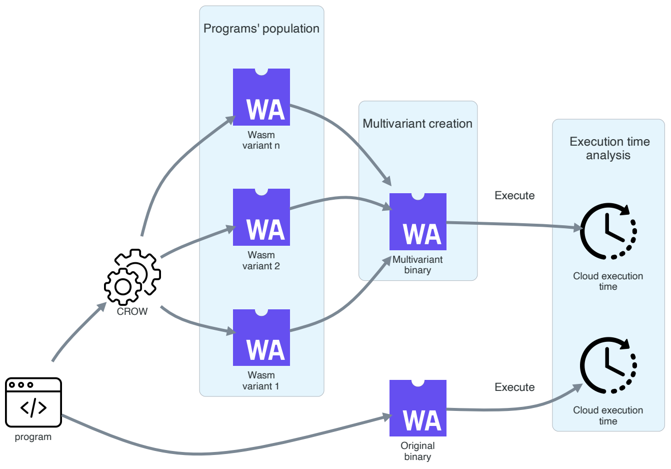


Figure 4.3: Multivariant binary creation and workflow for RQ3 answering.

In RQ3, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant binaries. We build multivariant binaries (according to Definition 1), and we deploy and execute them at the Edge. The usage of edge-cloud computing platforms to answer RQ3 is motivated by two reasons. First, it is an emerging technology. Using Wasm as an intermediate layer is better in terms of startup and memory usage, than containerization or virtualization [23, 39]. This has encouraged edge computing platforms like Cloudflare and Fastly to use Wasm to deploy client applications in a modular and sandboxed manner [36, 40]. Second, edge-cloud computing platforms are shown to not be completely secure [7] and multivariant execution offers a preemptive technique against predictable behaviors such as execution time.

■ Metrics

To answer RQ3, we build multivariant Wasm binaries (see Definition 1) meant to provide execution path randomization. We use the execution time of the multivariant binaries to answer RQ3. We use the same metric defined in Metric 6 for the execution time of multivariant binaries.

■ Protocol

We answer RQ3 by analyzing real-world scenarios on the Edge. Edge applications are designed to be deployed as isolated HTTP services, having one single responsibility that is executed at every HTTP request. This development model is known as serverless computing, or function-as-a-service [21, 7]. We deploy and execute the multivariant binaries as end-to-end HTTP services on the Edge, and we collect their execution times. To remove the natural jitter in the network, the execution times are measured at the backend space, *i.e.,* we collect the execution times inside the Edge node and not from the client computer. Therefore, we instrument the binaries to return the execution time as an HTTP header.

We do the collection of the execution times twice, for the original program and its multivariant binary. We deploy and execute the original and the multivariant binaries on 64 edge nodes located around the world. In Figure 4.4 we illustrate the world wide location of the edges nodes.

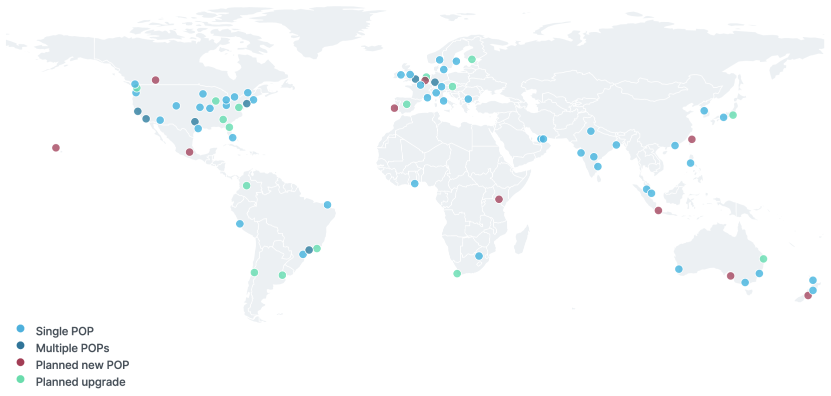


Figure 4.4: Screenshot taken from the Fastly Inc. platform used in our experiments for RQ3. Blue and darker blue dots represent the edge nodes used in our experiments.

We collect 100k execution times for each binary, both the original and multivariant binaries. The number of execution time samples is motivated by the seminal work of Morgan et al. [51]. We perform a Mann-Withney U test [101] to compare both execution-time distributions. If the P-value is lower than 0.05, the two compared distributions are different.

■ Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of our approach to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through dynamic analysis. We propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Besides, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for Wasm programs. In the next chapter, we present and discuss the results obtained with this methodology.

05 RESULTS

In this chapter, we sum up the results of the research of this thesis. We illustrate the key insights and challenges faced in answering each research question. To obtain our results, we followed the methodology formulated in Chapter 4.

■ 5.1 RQ_1. To what extent can we artificially generate program variants for WebAssembly?

As we describe in Section 4.2, our first research question aims to answer how to artificially generate Wasm program variants. This section is organized as follows. First we present the general results calculating the *Corpus population size* (Metric 3) and *Corpus unique population size* (Metric 4) for each corpus. Second, we discuss the challenges and limitations in program variants generation. Finally, we illustrate the most common code transformations performed by our approach and answer RQ1.

■ 5.1.1 Program's population

We summarize the results in Table 5.1. The table illustrates the corpus name, the number of functions to diversify, the number of successfully diversified functions (functions with at least one artificially created variant), the cumulative number of variants (*Corpus population size*) and the cumulative number of unique variants (*Corpus unique population size*).

We produce at least one unique program variant for 239/303 single function programs for Rosetta with one hour for a diversification timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of Libsodium and QrCode, we produce variants for 85/869 and 32/1849 functions respectively, with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants. For all programs in all corpora, we achieve 356/3021 successfully diversified functions, representing a 11.78% of the total. As the four and fifth columns show, the number of artificially created variants and the number of unique variants are larger than the original number of functions by one order of magnitude. In the case of Rosetta, the corpus population size is close to one million of programs. The remarkable difference between the total number of variants and the number of unique variants (fourth and fifth columns) is mainly due to the *replacements combining* process discussed in Section 3.2.

Corpus	#Functions	# Diversified	# Variants	# Unique Variants
Rosetta	303	239	809900	2678
Libsodium	869	85	4272	3805
QrCode	1849	32	6369	3314
	3021	356	820541	9797

Table 5.1: General program's populations statistics. The table is composed by the name of the corpus, the number of functions, the number of successfully diversified functions, the cumulative number of generated variants and the cumulative number of unique variants.

■ 5.1.2 Challenges for automatic diversification

We have observed a noticeable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions (third column of Table 5.1). Our approach successfully diversified 239/303, 85/869 and 32/1849 of the original functions for Rosetta, Libsodium and QrCode respectively.

We have noticed a remarkable difference between the number of diversified functions for each corpus, 809900 programs for Rosetta 4272 for Libsodium and 6369 for QrCode. The corpus population size for Rosetta is two orders of magnitude larger compared to the other two corpora. The reason behind the large number of variants for Rosetta is that, after a certain time, our approach starts to combine the code replacements to generate new variants. However, looking at the fifth column, the number of unique variants have the same order of magnitude for all corpora. The variants generated out of the combination of several code replacements are not necessarily unique. Some code replacements can dominate over others, generating the same Wasm programs.

A low timeout offers more unique variants compared to the population size despite the low number of diversified functions, like the Libsodium and QrCode cases. This happens because, CROW first generates variants out of single code replacements and then starts to combine them. Thus, more unique variants are generated in the very first moments of the diversification process with CROW.

Apart from the timeout and the combination of variants phenomenon, we manually analyze programs, searching for properties attempting to the generation of program variants using CROW. As we previously mentioned in Section 3.2, *constant inferring* is a new contribution of ours to the collection of Software Diversification strategies enumerated in Section 2.2. We have observed that our approach searches for a constant inferring for more than 45% of the instructions of each function while constant values cannot be inferred in all cases. The main reason is that memory operations are also included into the inferring while our tool is oblivious to a memory model, making unsuccessful the constant replacement.

5.1.3 Properties for large diversification

We manually analyzed the programs to study the critical properties of programs producing a high number of variants. This reveals one key factor that favors many unique variants: the presence of bounded loops. In these cases, we synthesize variants for the loops by replacing them with a constant, if the constant inferring is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which we can successfully infer constants.

A second key factor for synthesizing many variants relates to the presence of arithmetic expressions. The synthesis engine used by our approach, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, we generate unique variants by replacing multiplications with additions or shift left instructions (Listing 5.1). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 5.2). Besides, our implementation can use overflow and underflow of integers to produce variants (Listing 5.3).

Listing 5.1: Diversification through arithmetic expression replacement.

```
local.get 0       local.get 0
i32.const 2       i32.const 1
i32.mul           i32.shl
```

Listing 5.2: Diversification through inversion of comparison operations.

```
local.get 0       i32.const 11
i32.const 10      local.get 0
i32.gt_s          i32.le_s
```

Listing 5.3: Diversification through overflow of integer operands.

```
i32.const 2       i32.const 2
i32.mul           i32.mul
                  i32.const
                    -2147483647
                  i32.mul
```

At the Wasm level, we have not observed variants performing changes in the control flow structure of the program (S3). Yet, this is not the case when we manually analyze the machine code generated by V8 (as it was discussed in Section 2.1). For the generated machine code, we have observed that, for different variants, we are changing the number of jumps and its locations. The control flow change strategy (S3) is correctly achieved as a consequence of latter compilation of Wasm program variants.

Answer to RQ_1. To what extent can we artificially generate program variants for WebAssembly?

We can provide diversification for 11.78% of the programs in our corpora. Constant inferring, complemented with the high presence of arithmetic operations and bounded loops in the original program increased the number of program variants. Our method based on the inclusion of a diversifier in the LLVM pipeline proved to be feasible, by providing statically different Wasm variants.

■ 5.2 RQ_2. To what extent are the generated variants dynamically different?

Our second research question investigates the differences between program variants at runtime. To answer RQ2, we execute each program/variant generated to answer RQ1 for Rosetta corpus to collect their execution traces and execution times. For each programs' population we compare the stack operation traces (Metric 5) and the execution-time distributions (Metric 6) for each program/variant pair.

This section is organized as follows. First, we analyze the programs' populations by comparing the traces for each pair of program/variant with TraceDiff of Metric 5. The pairwise comparison will hint at the results at the population level. We analyze not only the differences of a variant regarding its original program, we also compare the variants against other variants. Second, we do the same pairwise strategy for the execution-time distributions Metric 6, performing a Mann-Withney U test for each pair of program/variant times distribution. Finally, we conclude and answer RQ2.

■ 5.2.1 Stack operation traces.

In Figure 5.1 we plot the distribution of all comparisons (in logarithmic scale) of all pairs of program/variant in each programs' population. All compared programs are statically different. Each vertical group of blue dots represents all the pairwise comparison of the traces (Metric 5) for a program of the Rosetta corpus for which we generate variants. Each dot represents a comparison between two programs' traces according to Metric 5. The programs are sorted by their number of variants in descending order. For the sake of illustration, we filter out those programs for which we generate only 2 unique variants.

We have observed that in the majority of the cases, the mean of the comparison values is remarkably large. We analyze the length of the traces, and one reason behind such large values of TraceDiff is that some variants result from constant inferring. For example, if a loop is replaced by a constant, instead of several symbols

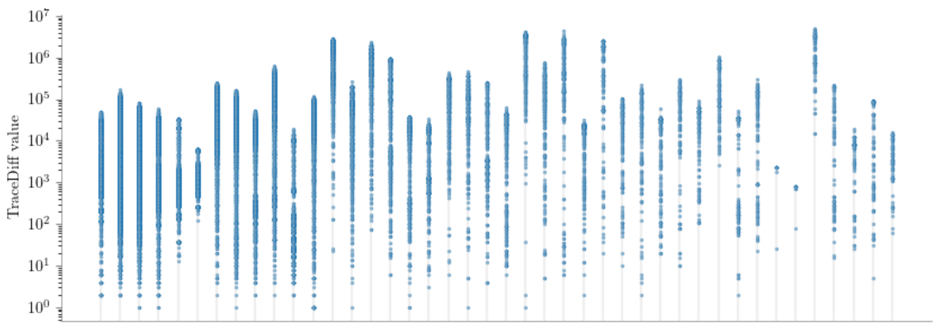


Figure 5.1: Pairwise comparison of programs' population traces in logarithmic scale. Each vertical group of blue dots represents a programs' population. Each dot represents a comparison between two program execution traces according to Metric 5.

in the stack operation trace, we observe one. Consequently, the distance between two program traces is significant.

In some cases, we have observed variants that are statically different for which TraceDiff value is zero, *i.e.,* they result in the same stack operation trace. We identified two main reasons behind this phenomenon. First, the code transformation that generates the variant targets a non-executed or dead code. Second, some variants have two different instructions that trigger the same stack operations. For example, the code replacements below illustrate the case.

```
(1) i32.lt_u          i32.lt_s          (3) i32.ne                 i32.lt_u
(2) i32.le_s          i32.lt_u          (4) local.get 6            local.get 4
```

In the four cases, the operators are different (original in gray color and the replacement in green color) leaving the same values for equal operands. The (1) and (2) cases are comparison operations leaving the value 0 or 1 in the stack taking into account the sign of their operands. In the third case, the replacement is less restricted to the original operator, but in both cases, the codes leave the same value in the stack. In the last case, both operands load a value of a local variable in the stack, the index of the local variable is different but the value of the variable that is appended to the trace is the same in both cases.

■ 5.2.2 Execution times.

Even when two programs of the same population offer different execution traces, their execution times can be similar (statistically speaking). In practice, the execution traces of Wasm programs are not necessarily accessible, being not the case with the execution time. For example, in our current experimentation we need to use our own instrumentation of the execution engine to collect the stack trace operations while the execution time is naturally accessible in any execution environment. This mentioned reasoning enforces our comparison of the execution times for the generated variants. Besides the execution times of programs can be used by malicious clients to construct personalized attacks [51]. Therefore, by measuring the execution times, we assess the diversification of observable behaviors evaluated in real-world security scenarios.

For each program's population, we compare the execution-time distributions, Metric 6, of each pair of program/variant. Overall diversified programs, 169 out of 239 (71%) have at least one variant with a different execution-time distribution than the original program (P-value < 0.01 in the Mann-Withney test). This result shows that we effectively generate variants that yield significantly different execution times.

By analyzing the data, we observe the following trends. First, if our tool infers control-flows as constants in the original program, the variants execute faster than the original, sometimes by one order of magnitude. On the other hand, if the code is augmented with more instructions, the variants tend to run slower than the original.

In both cases, we generate a variant with a different execution time than the original. Both cases are good from a randomization perspective since this minimizes the certainty a malicious user can have about the program's behavior. Therefore, a deeper analysis of how this phenomenon can be used to enforce security will be discussed in answering RQ3.

To better illustrate the differences between executions times in the variants, we dissect the execution-time distributions for one programs' population of Rosetta. The plots in Figure 5.2 show the execution-time distributions for the `Hilbert curve` program and their variants. We illustrate time diversification with this program because, we generate unique variants with all types of transformations previously discussed in Section 5.1. In the plots along the X-axis, each vertical set of points represents the distribution of 100000 execution times per program/variant. The Y-axis represents the execution time value in milliseconds. The original program is highlighted in green color: the distribution of 10000 execution times is given on the left-most part of the plot, and its median execution time is represented as a horizontal dashed line. The median execution time is represented as a blue dot for each execution-time distribution, and the vertical gray lines represent the entire distribution. The bolder gray line represents the 75% interquartile. The program variants are sorted concerning the median execution time in descending order.

For the illustrated program, many diversified variants are optimizations (blue dots below the green bar). The last third represents faster variants resulting from code transformations that optimize the original program. Our tool provides program variants in the whole spectrum of time executions, lower and faster variants than the original program. The developer is in charge of deciding between taking all variants or only the ones providing the same or less execution time for the

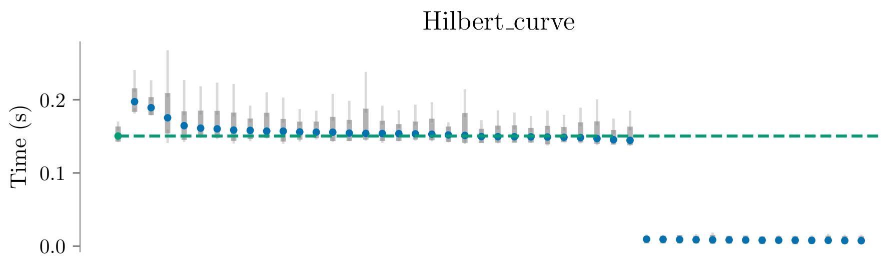


Figure 5.2: Execution-time distributions for `Hilber_curve` program and its variants. Baseline execution time mean is highlighted with the green horizontal line.

sake of performance. Nevertheless, this result calls for using this timing spectrum phenomenon to provide binaries with unpredictable execution times by combining variants. The feasibility of this idea will be discussed in Section 5.3.

Answer to RQ_2. To what extent are the generated variants dynamically different?

We empirically demonstrate that our approach generates program variants for which execution traces are different. We stress the importance of complementing static and dynamic studies of programs variants. For example, if two programs are statically different, that does not necessarily mean different runtime behavior. There is at least one generated variant for all executed programs that provides a different execution trace. We generate variants that exhibit a significant diversity of execution times. Concretely, for 169/239 (71%) of the diversified programs, at least one variant has an execution-time distribution that is different compared to the execution-time distribution of the original program. The result from this study encourages the composition of the variants to provide a resilient execution.

■ 5.3 RQ_3. To what extent do the artificial variants exhibit different execution times on edge-cloud platforms?

Here we investigate the impact of the composition of program variants into multivariant binaries. To answer this research question, we create multivariant binaries from the program variants generated for Libsodium and QrCode corpora. Then, we deploy the multivariant binaries into the Edge and collect their execution times.

■ 5.3.1 Execution times

We compare the execution-time distributions for each program for the original and the multivariant binary. All distributions are measured on 100k executions of the program along all Edge platform nodes. We have observed that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution-time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 5.3, each subplot represents the quantile-quantile plot [100] of the two distributions, original and multivariant binary. This kind of plots is used to compare the shapes of distributions, providing a graphical comparison of location, scale, and skewness for two distributions. The dashed line cutting the subplot represents the case in which the two distributions are equal, *i.e.,* for two equal distribution we would have all blue dots over the dashed line. These plots reveal that the execution times are different and are spread over a more extensive range of values than the original binary. The standard deviation of the execution time values evidences the latter, the original binaries have lower values while the multivariant binaries have higher values up to 100 times the original. Besides, this can be graphically appreciated in the plots when the blue dots cross the reference line from the bottom of the dashed line to the top. This is evidence that execution time is less predictable for multivariant binaries than original ones. This phenomenon is present because the choice of function variants is randomized at each function invocation, and the variants have different execution times due to the code transformations, i.e., some variants execute more instructions than others.

Answer to RQ_3. To what extent do the artificial variants exhibit different execution times on edge-cloud platforms?

The execution-time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. The distribution for the multivariant binary is different and even more spread than the original one. Consequently, attacks relying on measuring precise execution times [51] of a function are made a lot harder to conduct.

■ Conclusions

Our approach introduces static and dynamic, variants for up to 11.78% of the programs in our three corpora, increasing the original count of unique programs by 3.21 times (9797/3021). We highlighted the importance of complementing static and dynamic studies for programs diversification. Our results on the study of the

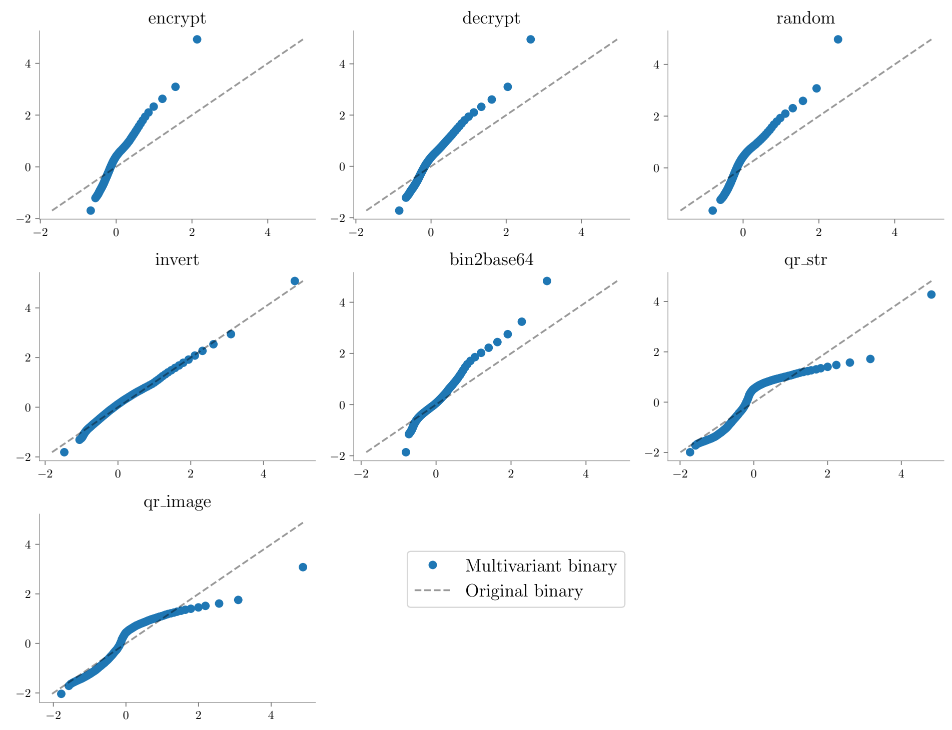


Figure 5.3: Execution-time distributions. Each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary.

execution of the generated variants encourages the composition of the variants to provide a resilient execution. Moreover, combining function variants in multivariant binaries makes virtually impossible to predict which variant is executed for a given query. We empirically demonstrate the feasibility and the application of automatically generating Wasm program variants.

06 CONCLUSION AND FUTURE WORK

Wasm has become a new technology for web browsers and standalone engines such as the ones used in edge-cloud platforms. Wasm is designed with security and sandboxing premises, yet, is still vulnerable. Besides, since it is a relatively new technology, new vulnerabilities appear in the wild faster than the adoption of patches and defenses. As a widely studied field, software diversification could be a solution for known and yet-unknown vulnerabilities. Yet, there is no research on this field for Wasm.

We propose an automatic approach to generate software diversification for Wasm in this work. In addition, we provide complementary implementation for our approaches, including a generic LLVM superdiversifier that potentially extends our ideas to other programming languages. We empirically demonstrate the impact of our approach by providing Randomization and Multivariant Execution (MVE) for Wasm. For this, we provide two tools, CROW and MEWE. CROW completely automatizes the process by using a superdiversifier. Besides, MEWE provides execution path randomization for an MVE. This chapter is organized into two sections. In Section 6.1, we summarize the main results we found by answering our research questions enunciated in Chapter 1. Finally, Section 6.2 describes potential future work that could extend this dissertation.

■ 6.1 Summary of the results

We enunciate the three research questions in Chapter 1. With the first research question, we investigate the feasibility of generating software diversification for Wasm through the engineering of the LLVM pipeline. Besides, we highlight static properties of the software diversification for Wasm generated by our approaches. The generated variants are semantically equivalent to their respective original programs. We study the properties of the generated variants at the level of generated programs' population. Thus, we identify the challenges that attempt against the generation of unique program variants. Besides, we highlight the code properties that offer numerous program variants. We answer our first research question by creating near 1 million program variants for 3021 original programs. With CROW, we create program variants for the 11.78% of the programs in our corpora.

Complementing our first research question, we evaluate the dynamic properties of the program variants generated to answer our first research question. We execute each of the 303 original programs and its generated variants for the Rosetta. For each execution, we collect their execution trace and their execution times. We demonstrate that the Wasm variants generated by CROW offer remarkably different execution traces. Similarly, the execution times are different between each program and its variants. For the 71% of the diversified programs, at least one variant has an execution-time distribution different from the original program's execution time distribution. Moreover, CROW generates both faster and slower variants enforcing its usage for multivariant execution environments. In addition, we highlighted the importance of dynamic analysis for software diversification.

Our last and third research question evaluates the impact of providing a worldwide MVE for Wasm. We use MEWE to build multivariant binaries for the program variants generated for Libsodium and QrCode corpora. We collect their execution times by deploying the generated multivariant binaries in a production-based edge-cloud platform. The addition of runtime path randomization to multivariant binaries provides significant differences between the execution of the original binary and the multivariant binary. The observed differences lead us to conclude that no specific variant can be inferred from studying the execution time of the multivariant binaries. Therefore, attacks that rely on measuring precise execution times are more challenging to conduct.

Overall, these results show that our approaches can provide an automated end-to-end solution for diversifying Wasm programs. Our approaches harden observable properties commonly used to conduct attacks, such as static code analysis, execution traces, and execution time. Therefore, our approaches harden Wasm against unknown and yet-unknown vulnerabilities. Remarkably, we provide a generic LLVM superdiversifier that potentially extends our ideas to other programming languages.

■ 6.2 Future work

There are many directions in which software diversification for Wasm could be researched further. In this section, we describe three possible orthogonal lines of work.

CROW and MEWE: Along with this dissertation, we highlighted challenges and limitations. In all cases, we proposed solutions, yet, some of them could be explored more in-depth. As we mentioned in Section 5.1 our solution provides program variants but remarkably lower unique variants as a consequence of the replacement combining process of CROW (Section 3.2). Techniques relying on intelligent heuristics could help increase the generation of unique variants by early discarding unsound combinations. On the other hand, constant inferring does not always finish in a successful replacement due to the CROW's obliviousness to some computation models, such as memory operations. A solution could also be to use heuristics to select which part of the code is more probable to become a constant inferred assignment. On the other hand, MEWE introduces overhead during the execution of the multivariant binaries. We identified the dispatcher calling the function variants as the main reason. Each time a new variant executes, it involves the introduction of a new function call through the dispatcher. Our variants are artificially created. Thus, their bodies could be directly inlined in the dispatcher's body. This means that we can reduce the number of function calls by inlining the variant. Nevertheless, a deeper study on the security consequences is needed.

Obfuscation, data augmentation and malware classification: Wasm is extensively used for cryptocurrency mining. Sometimes the crypto-mining is done without the consent of users, creating what is called crypto-malwares [11]. Antivirus software could detect them. However, a recent work [4] shows that malware classifiers could be bypassed with the correct obfuscation technique. Our diversification approach could be used to increase resilience in malware classifiers by training them with augmented datasets on semantically equivalent malwares. On the other hand, superoptimization can be used to build a canonical code representation of a variant's population. Therefore, if a classifier uses a canonical representation, then malware obfuscation could be mitigated.

Better fuzzing: Fuzzers have become one of the most used techniques for automated testing [43], and compilers are not the exception. Fastly uses this technique to test their compiler, Lucet. Their fuzzing technique randomly creates different Wasm binaries and passes them to the compiler. If the compiler crashes, a bug report is created and fixed later. Our approaches created one binary that crashed their compiler [16], after they had no bug for months. Therefore, our code transformations outperform their code generation for testing. This highlighted the need for better strategies for stressing compilers, interpreters, and validators for Wasm. CROW and MEWE might be used for fuzzing, preventing vulnerabilities, and providing better testing of systems.

BIBLIOGRAPHY

[1] Stiévenart,Q., De Roover,C., and Ghafari,M. (2022). Security risks of porting c programs to webassembly. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1713–1722, New York, NY, USA. Association for Computing Machinery.

[2] Romano,A., Lehmann,D., Pradel,M., and Wang,W. (2022). Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1101–1116, Los Alamitos, CA, USA. IEEE Computer Society.

[3] Harrand,N. (2022). *Software Diversity for Third-Party Dependencies*. PhD thesis, KTH, Software and Computer systems, SCS. QCR 20220413.

[4] Bhansali,S., Aris,A., Acar,A., Oz,H., and Uluagac,A. S. (2022). A first look at code obfuscation for webassembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '22, page 140–145, New York, NY, USA. Association for Computing Machinery.

[5] Voulimeneas,A., Song,D., Larsen,P., Franz,M., and Volckaert,S. (2021). dmvx: Secure and efficient multi-variant execution in a distributed setting. In *Proceedings of the 14th European Workshop on Systems Security*, pages 41–47.

[6] Spies,B. and Mock,M. (2021). An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10.

[7] Narayan,S., Disselkoen,C., Moghimi,D., Cauligi,S., Johnson,E., Gang,Z., Vahldiek-Oberwagner,A., Sahita,R., Shacham,H., Tullsen,D., et al. (2021). Swivel: Hardening webassembly against spectre. In *USENIX Security Symposium*.

[8] Lee,S., Kang,H., Jang,J., and Kang,B. B. (2021). Savior: Thwarting stack-based memory safety violations by randomizing stack layout. *IEEE Transactions on Dependable and Secure Computing*.

[9] Ko,Y., Rezk,T., and Serrano,M. (2021). Securejs compiler: Portable memory isolation in javascript. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1265–1274, New York, NY, USA. Association for Computing Machinery.

[10] Johnson,E., Thien,D., Alhessi,Y., Narayan,S., Brown,F., Lerner,S., McMullen,T., Savage,S., and Stefan,D. (2021). Sfi safety for native-compiled wasm. *NDSS. Internet Society*.

[11] Hilbig,A., Lehmann,D., and Pradel,M. (2021). An empirical study of real-world webassembly binaries: Security, languages, use cases. *Proceedings of the Web Conference 2021*.

[12] Garcés,L., Martínez-Fernández,S., Oliveira,L., Valle,P., Ayala,C., Franch,X., and Nakagawa,E. Y. (2021). Three decades of software reference architectures: A systematic mapping study. *Journal of Systems and Software*, 179:111004.

[13] Cabrera Arteaga,J., Laperdrix,P., Monperrus,M., and Baudry,B. (2021). Multi-Variant Execution at the Edge. *arXiv e-prints*, page arXiv:2108.08125.

[14] Cabrera Arteaga,J., Floros,O., Vera Perez,O., Baudry,B., and Monperrus,M. (2021). Crow: code diversification for webassembly. In *MADWeb, NDSS 2021*.

[15] (2021). Webassembly system interface. `https://github.com/WebAssembly/WASI`.

[16] (2021). Stop a wasm compiler bug before it becomes a problem | fastly. `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`.

[17] (2021). Global CDN Disruption. `https://www.fastly.com/blog/summary-of-june-8-outage`.

[18] Xu,Y., Solihin,Y., and Shen,X. (2020). Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proc. of ASPLOS*, pages 987–1000.

[19] Wen,E. and Weber,G. (2020). Wasmachine: Bring iot up to speed with a webassembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–4. IEEE.

[20] Tsoupidi,R. M., Lozano,R. C., and Baudry,B. (2020). Constraint-based software diversification for efficient mitigation of code-reuse attacks. *ArXiv*, abs/2007.08955.

[21] Shillaker,S. and Pietzuch,P. (2020). Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference*, pages 419–433.

[22] Runeson,P., Engström,E., and Storey,M.-A. (2020). *The Design Science Paradigm as a Frame for Empirical Software Engineering*, pages 127–147. Springer International Publishing, Cham.

[23] Mendki,P. (2020). Evaluating webassembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*, pages 161–166.

[24] Lehmann,D., Kinder,J., and Pradel,M. (2020). Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[25] Harrand,N., Soto-Valero,C., Monperrus,M., and Baudry,B. (2020). Java decompiler diversity and its application to meta-decompilation. *Journal of Systems and Software*, 168:110645.

[26] Gadepalli,P. K., McBride,S., Peach,G., Cherkasova,L., and Parmer,G. (2020). Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, page 265–279.

[27] Chen,D. and W3C group (2020). WebAssembly documentation: Security. `https://webassembly.org/docs/security/`. Accessed: 18 June 2020.

[28] Cabrera Arteaga,J., Donde,S., Gu,J., Floros,O., Satabin,L., Baudry,B., and Monperrus,M. (2020). *Superoptimization of WebAssembly Bytecode*, page 36–40. Association for Computing Machinery, New York, NY, USA.

[29] Bryant,D. (2020). Webassembly outside the browser: A new foundation for pervasive computing. In *Proc. of ICWE 2020*, pages 9–12.

[30] Österlund,S., Koning,K., Olivier,P., Barbalace,A., Bos,H., and Giuffrida,C. (2019). kmvx: Detecting kernel information leaks with multi-variant execution. In *ASPLOS*.

[31] Gurdeep Singh,R. and Scholliers,C. (2019). Warduino: A dynamic webassembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, pages 27–36, New York, NY, USA. ACM.

[32] Churchill,B., Padon,O., Sharma,R., and Aiken,A. (2019). Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA. Association for Computing Machinery.

[33] Cabrera Arteaga,J., Monperrus,M., and Baudry,B. (2019). Scalable comparison of javascript v8 bytecode traces. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2019, page 22–31, New York, NY, USA. Association for Computing Machinery.

[34] Bytecode Alliance (2019). Bytecode Alliance. `https://bytecodealliance.org/`.

[35] Aga,M. T. and Austin,T. (2019). Smokestack: thwarting dop attacks with runtime stack layout randomization. In *Proc. of CGO*, pages 26–36.

[36] Varda,K. (2018). Webassembly on cloudflare workers. Technical report.

[37] Lu,K., Xu,M., Song,C., Kim,T., and Lee,W. (2018). Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing.*

[38] Li,J., Zhao,B., and Zhang,C. (2018). Fuzzing: a survey. *Cybersecurity*, 1(1):1–13.

[39] Jacobsson,M. and Wåhslén,J. (2018). Virtual machine execution for wearables based on webassembly. In *EAI International Conference on Body Area Networks*, pages 381–389. Springer, Cham.

[40] Hickey,P. (2018). Announcing lucet: Fastly's native webassembly compiler and runtime. Technical report.

[41] Genkin,D., Pachmanov,L., Tromer,E., and Yarom,Y. (2018). Drive-by key-extraction cache attacks from portable code. *IACR Cryptol. ePrint Arch.*, 2018:119.

[42] Belleville,N., Couroussé,D., Heydemann,K., and Charles,H.-P. (2018). Automated software protection for the masses against side-channel attacks. *ACM Trans. Archit. Code Optim.*, 15(4).

[43] Zalewski,M. (2017). American fuzzy lop.

[44] WebAssembly Community Group (2017). WebAssembly Specification. `https://webassembly.github.io/spec/core/syntax/index.html`.

[45] Sasnauskas,R., Chen,Y., Collingbourne,P., Ketema,J., Lup,G., Taneja,J., and Regehr,J. (2017). Souper: A Synthesizing Superoptimizer. *arXiv preprint 1711.04422*.

[46] Oracle (2017). JDK 9 Release Notes. Deprecation of Java Applets. `https://www.oracle.com/java/technologies/javase/9-deprecated-features.html`.

[47] Haas,A., Rossberg,A., Schuff,D. L., Schuff,D. L., Titzer,B. L., Holman,M., Gohman,D., Wagner,L., Zakai,A., and Bastien,J. F. (2017). Bringing the web up to speed with webassembly. *PLDI*.

[48] Van Es,N., Nicolay,J., Stievenart,Q., D'Hondt,T., and De Roover,C. (2016). A performant scheme interpreter in asm.js. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1944–1951, New York, NY, USA. Association for Computing Machinery.

[49] Phothilimthana,P. M., Thakur,A., Bodik,R., and Dhurjati,D. (2016). Scaling up superoptimization. *SIGARCH Comput. Archit. News*, 44(2):297–310.

[50] Couroussé,D., Barry,T., Robisson,B., Jaillon,P., Potin,O., and Lanet,J.-L. (2016). Runtime code polymorphism as a protection against side channel attacks. In *IFIP International Conference on Information Security Theory and Practice*, pages 136–152. Springer.

[51] Morgan,T. D. and Morgan,J. W. (2015). Web timing attacks made practical. *Black Hat*.

[52] Davi,L., Liebchen,C., Sadeghi,A.-R., Snow,K. Z., and Monrose,F. (2015). Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*.

[53] Crane,S., Homescu,A., Brunthaler,S., Larsen,P., and Franz,M. (2015). Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, pages 8–11.

[54] Baudry,B. and Monperrus,M. (2015). The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Comput. Surv.*, 48(1).

[55] Alon Zakai (2015). asm.js Speedups Everywhere. `https://hacks.mozilla.org/2015/03/asm-speedups-everywhere/`.

[56] Agosta,G., Barenghi,A., Pelosi,G., and Scandale,M. (2015). The MEET approach: Securing cryptographic embedded software against side channel attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1320–1333.

[57] Zakai and colleagues (2014b). Emscripten. `https://emscripten.org/`.

[58] Zakai and colleagues (2014a). asm.js. `http://asmjs.org/spec/latest/`.

[59] Le,V., Afshari,M., and Su,Z. (2014). Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226.

[60] Okhravi,H., Rabe,M., Mayberry,T., Leonard,W., Hobson,T., Bigelow,D., and Streilein,W. (2013). Survey of cyber moving targets. *Massachusetts Inst of Technology Lexington Lincoln Lab, No. MIT/LL-TR-1166.*

[61] Mulazzani,M., Reschl,P., Huber,M., Leithner,M., Schrittwieser,S., Weippl,E., and Wien,F. (2013). Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, volume 5, page 4. Citeseer.

[62] Homescu,A., Neisius,S., Larsen,P., Brunthaler,S., and Franz,M. (2013). Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE.

[63] Jackson,T. (2012). *On the Design, Implications, and Effects of Implementing Software Diversity for Security.* PhD thesis, University of California, Irvine.

[64] Cleemput,J. V., Coppens,B., and De Sutter,B. (2012). Compiler mitigations for time attacks on modern x86 processors. *ACM Trans. Archit. Code Optim.*, 8(4).

[65] Sidiroglou-Douskos,S., Misailovic,S., Hoffmann,H., and Rinard,M. (2011). Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 124–134, New York, NY, USA. Association for Computing Machinery.

[66] Jackson,T., Salamat,B., Homescu,A., Manivannan,K., Wagner,G., Gal,A., Brunthaler,S., Wimmer,C., and Franz,M. (2011). Compiler-generated software diversity. In *Moving Target Defense*, pages 77–98. Springer.

[67] Amarilli,A., Müller,S., Naccache,D., Page,D., Rauzy,P., and Tunstall,M. (2011). Can code polymorphism limit information leakage? In *IFIP International Workshop on Information Security Theory and Practices*, pages 1–21. Springer.

[68] Guha,A., Saftoiu,C., and Krishnamurthi,S. (2010). The essence of javascript. In D'Hondt,T., editor, *ECOOP 2010 – Object-Oriented Programming*, pages 126–150, Berlin, Heidelberg. Springer Berlin Heidelberg.

[69] Chen,T. Y., Kuo,F.-C., Merkel,R. G., and Tse,T. H. (2010). Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83:60–66.

[70] Salamat,B., Jackson,T., Gal,A., and Franz,M. (2009). Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46.

[71] Lala,J. H. and Schneider,F. B. (2009). It monoculture security risks and defenses. *IEEE Security & Privacy*, 7(1):12–13.

[72] Maia,M. D. A., Sobreira,V., Paixão,K. R., Amo,R. A. D., and Silva,I. R. (2008). Using a sequence alignment algorithm to identify specific and common code from execution traces. In *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*, pages 6–10.

[73] Jacob,M., Jakubowski,M. H., Naldurg,P., Saw,C. W. N., and Venkatesan,R. (2008). The superdiversifier: Peephole individualization for software protection. In *International Workshop on Security*, pages 100–120. Springer.

[74] de Moura,L. and Bjørner,N. (2008). Z3: An efficient smt solver. In Ramakrishnan,C. R. and Rehof,J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[75] Yu,D., Chander,A., Islam,N., and Serikov,I. (2007). Javascript instrumentation for browser security. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 237–249, New York, NY, USA. Association for Computing Machinery.

[76] Salamat,B., Gal,A., Jackson,T., Manivannan,K., Wagner,G., and Franz,M. (2007). Stopping buffer overflow attacks at run-time: Simultaneous multi-variant program execution on a multicore processor. Technical report, Technical Report 07-13, School of Information and Computer Sciences, UCIrvine.

[77] Microsoft (2007). Silverlight. `https://www.microsoft.com/silverlight/`.

[78] Bruschi,D., Cavallaro,L., and Lanzi,A. (2007). Diversified process replicæ for defeating memory error exploits. In *Proc. of the Int. Performance, Computing, and Communications Conference*.

[79] Younan,Y., Pozza,D., Piessens,F., and Joosen,W. (2006). Extended protection against stack smashing attacks without performance loss. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 429–438.

[80] Cox,B., Evans,D., Filipi,A., Rowanhill,J., Hu,W., Davidson,J., Knight,J., Nguyen-Tuong,A., and Hiser,J. (2006). N-variant systems: a secretless framework for security through diversity. In *Proc. of USENIX Security Symposium*, USENIX-SS'06.

[81] Pohl,K., Böckle,G., and Van Der Linden,F. (2005). *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer.

[82] Grosskurth,A. and Godfrey,M. W. (2005). A reference architecture for web browsers. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 661–664. IEEE.

[83] Bhatkar,S., Sekar,R., and DuVarney,D. C. (2005). Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*, pages 271–286.

[84] El-Khalil,R. and Keromytis,A. D. (2004). Hydan: Hiding information in program binaries. In Lopez,J., Qing,S., and Okamoto,E., editors, *Information and Communications Security*, pages 187–199, Berlin, Heidelberg. Springer Berlin Heidelberg.

[85] LLVM (2003). The LLVM Compiler Infrastructure . `https://llvm.org/`.

[86] Kc,G. S., Keromytis,A. D., and Prevelakis,V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proc. of CCS*, pages 272–280.

[87] Goth,G. (2003). Addressing the monoculture. *IEEE Security & Privacy*, 1(06):8–10.

[88] Bhatkar,S., DuVarney,D. C., and Sekar,R. (2003). Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium.*

[89] Barrantes,E. G., Ackley,D. H., Forrest,S., Palmer,T. S., Stefanovic,D., and Zovi,D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. CCS*, pages 281–289.

[90] Chew,M. and Song,D. (2002). Mitigating buffer overflows by operating system randomization. Technical Report CS-02-197, Carnegie Mellon University.

[91] Forrest,S., Somayaji,A., and Ackley,D. (1997). Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, pages 67–72.

[92] Microsoft (1996). Microsoft Announces ActiveX Technologies. `https://web.archive.org/web/20090828024117/http://www.microsoft.com/presspass/press/1996/mar96/activxpr.mspx`.

[93] Cohen,F. B. (1993). Operating system protection through program evolution. *Computers & Security*, 12(6):565–584.

[94] Tim Berners-Lee (1990). The WorldWideWeb browser. `https://www.w3.org/People/Berners-Lee/WorldWideWeb.html`.

[95] Pettis,K. and Hansen,R. C. (1990). Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 16–27, New York, NY, USA. Association for Computing Machinery.

[96] Henry,M. (1987). Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126.

[97] Avizienis and Kelly (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80.

[98] Ryder,B. G. (1979). Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226.

[99] Needleman,S. B. and Wunsch,C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. 48(3):443–453.

[100] Gnanadesikan,R. and Wilk,M. B. (1968). Probability plotting methods for the analysis of data. *Biometrika*, 55(1):1–17.

[101] Mann,H. B. and Whitney,D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60.

[102] Cox,M. R. (1893). *Cinderella: Three hundred and forty-five variants of Cinderella, Catskin, and Cap o'Rushes.* Number 31. Folk-lore Society.

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus

Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

`https://doi.org/10.1145/3397537.3397567`

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus

Network and Distributed System Security Symposium (NDSS 2021), MADWeb

`https://doi.org/10.14722/madweb.2021.23004`

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry

Preprint version

Accepted at the Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)

preprint link `https://ui.adsabs.harvard.edu/abs/2021arXiv210808125C`

Multi-variant Execution at the Edge

JAVIER CABRERA-ARTEAGA, KTH Royal Institute of technology, Sweden
PIERRE LAPERDRIX, CNRS, France
MARTIN MONPERRUS, KTH Royal Institute of Technology, Sweden
BENOIT BAUDRY, KTH Royal Institute of Technology, Sweden

Edge-Cloud computing offloads parts of the computations that traditionally occurs in the cloud to edge nodes. The binary format WebAssembly is increasingly used to distribute and deploy services on such platforms. Edge-Cloud computing providers let their clients deploy stateless services in the form of WebAssembly binaries, which are then translated to machine code, sandboxed and executed at the edge. In this context, we propose a technique that (i) automatically diversifies WebAssembly binaries that are deployed to the edge and (ii) randomizes execution paths at runtime. Thus, an attacker cannot exploit all edge nodes with the same payload. Given a service, we automatically synthesize functionally equivalent variants for the functions providing the service. All the variants are then wrapped into a single multivariant WebAssembly binary. When the service endpoint is executed, every time a function is invoked, one of its variants is randomly selected. We implement this technique in the MEWE tool and we validate it with 7 services for which MEWE generates multivariant binaries that embed hundreds of function variants. We execute the multivariant binaries on the world-wide edge platform provided by Fastly, as part as a research collaboration. We show that multivariant binaries exhibit a real diversity of execution traces across the whole edge platform distributed around the globe.

Additional Key Words and Phrases: Diversification, Moving Target Defense, Edge-Cloud computing, Multivariant execution, WebAssembly.

ACM Reference Format:
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, and Benoit Baudry. 2022. Multi-variant Execution at the Edge. 1, 1 (August 2022), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Edge-Cloud computing distributes a part of the data and computation to edge nodes [20, 56]. Edge nodes are servers located in many countries and regions so that Internet resources get closer to the end users, in order to reduce latency and save bandwidth. Video and music streaming services, mobile games, as well as e-commerce and news sites leverage this new type of cloud architecture to increase the quality of their services. For example, the New York Times website was able to serve more than 2 million concurrent visitors during the 2016 US presidential election with no difficulty thanks to Edge computing [4].

The state of the art of edge computing platforms like Cloudflare or Fastly use the binary format WebAssembly (aka Wasm) [28, 57] to deploy and execute on edge nodes. WebAssembly is a portable bytecode format designed to be lightweight, fast and safe [17, 27]. After compiling code to a WebAssembly binary, developers spawn an edge-enabled compute service by deploying the binary on all nodes in an Edge platform. Thanks to its simple memory and computation model, WebAssembly is considered safe [44], yet is not exempt of vulnerabilities either at the execution engine's level [54] or the binary itself [38]. Implementations in both, browsers and standalone runtimes [45], have been found to be vulnerable [38, 45]. This means that if one node in an Edge network is vulnerable, all the others are vulnerable in the exact same manner. In other words, the same attacker payload would break all edge nodes at once [46]. This illustrates how Edge computing is fragile with respect to systemic vulnerabilities for the whole network, like it happened on June 8, 2021 for Fastly [3].

In this work, we introduce Multivariant Execution for WebAssembly in the Edge (MEWE), a framework that generates diversified WebAssembly binaries so that no two executions in the edge network are identical. Our solution is inspired by N-variant systems [23] where diverse variants are assembled for secretless security. Here, our goal is to drastically increase the effort for exploitation through large-scale execution path randomization. MEWE operates in two distinct steps. At compile time, MEWE generates *variants* for different functions in the program. A function variant is semantically identical to the original function but structurally different, i.e., binary instructions are in different orders or have been replaced with equivalent ones. All the function variants for one service are then embedded in a single multivariant WebAssembly binary. At runtime, every time a function is invoked, one of its variant is randomly selected. This way, the actual execution path taken to provide the service is randomized each time the service is executed, hardening Break-Once-Break-Everywhere (BOBE) attacks.

We experiment MEWE with 7 services, composed of hundreds of functions. We successfully synthesize thousands of function variants, which create orders of magnitude more possible execution paths than in the original service. To determine the runtime randomness of the embedded paths, we deploy and run the mutlivariant binaries on the Fastly edge computing platform (leading CDN platform). We collaborated with Fastly to experiment MEWE on the actual production edge computing nodes that they provide to their clients. This means that all our experiments ran in a real-world setting. For this experiment, we execute each multivariant binary several times on every edge computing node provided by Fastly. Our experiment shows that the multivariant binaries render the same service as the original, yet with highly diverse execution traces.

The novelty of our contribution is as follows. First, we are the first to perform software diversification in the context of edge computing,

Authors' addresses: Javier Cabrera-Arteaga, javierca@kth.se, KTH Royal Institute of technology, Sweden; Pierre Laperdrix, pierre.laperdrix@inria.fr, CNRS, France; Martin Monperrus, martin.monperrus@kth.se, KTH Royal Institute of Technology, Sweden; Benoit Baudry, baudry@kth.se, KTH Royal Institute of Technology, Sweden.

with experiments performed on a real-world, large-scale, commercial edge computing platform (Fastly). Second, very few works have looked at software diversity for WebAssembly [18, 45], our paper contributes to proving the feasibility of this challenging endeavour.

To sum up, our contributions are:

- MEWE: a framework that builds multivariant WebAssembly binaries for edge computing, combining the automatic synthesis of semantically equivalent function variants, with execution path randomization.
- Original results on the large-scale diversification of WebAssembly binaries, at the function and execution path levels.
- Empirical evidence of the feasibility of deploying our novel multivariant execution scheme on a real-world edge-computing platform.
- A publicly available prototype system, shared for future research on the topic: https://github.com/Jacarte/MEWE.

This work is structured as follows. First, Section 2 present a background on WebAssembly and its usage in an edge-cloud computing scenario. Section 3 introduces the architecture and foundation of MEWE while Section 4 and Section 5 present the different experiments we conducted to show the feasibility of our approach. Section 6 details the Related Work while Section 7 concludes this paper.

2 BACKGROUND

In this section we introduce WebAssembly, as well as the deployment model that edge-cloud platforms such as Fastly provide to their clients. This forms the technical context for our work.

2.1 WebAssembly

WebAssembly is a bytecode designed to bring safe, fast, portable and compact low-level code on the Web. The language was first publicly announced in 2015 and formalized by Haas et al. [27]. Since then, most major web browsers have implemented support for the standard. Besides the Web, WebAssembly is independent of any specific hardware, which means that it can run in standalone mode. This allows for the adoption of WebAssembly outside web browsers [17], e.g., for edge computing [45].

```c
int f(int x) {
    return 2 * x + x;
}
```

Listing 1. C function that calculates the quantity $2x + x$

```wasm
(module
  (type (;0;) (func (param i32) (result i32)))
  (func (;0;) (type 0) (param i32) (result i32)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
    (export "f" (func 0)))
```

Listing 2. WebAssembly code for Listing 1.

WebAssembly binaries are usually compiled from source code like C/C++ or Rust. Listing 1 and 2 illustrate an example of a C function turned into WebAssembly. Listing 1 presents the C code of one function and Listing 2 shows the result of compiling this function into a WebAssembly module. The WebAssembly code is further interpreted or compiled ahead of time into machine code.

2.2 Web Assembly and Edge Computing

Using Wasm as an intermediate layer is better in terms of startup and memory usage, than containerization or virtualization [32, 44]. This has encouraged edge computing platforms like Cloudflare or Fastly to adopt WebAssembly to deploy client applications in a modular and sandboxed manner [28, 57]. In addition, WebAssembly is a compact representation of code, which saves bandwidth when transporting code over the network .

Client applications that are designed to be deployed on edge-cloud computing platforms are usually isolated services, having one single responsibility. This development model is known as serverless computing, or function-as-a-service [45, 53]. The developers of a client application implement the isolated services in a given programming language. The source code and the HTTP harness of the service are then compiled to WebAssembly. When client application developers deploy a WebAssembly binary, it is sent to all edge nodes in the platform. Then, the WebAssembly binary is compiled on each node to machine code. Each binary is compiled in a way that ensures that the code runs inside an isolated sandbox.

2.3 Multivariant Execution

In 2006, security researchers of University of Virginia have laid the foundations of a novel approach to security that consists in executing multiple variants of the same program. They called this "N-variant systems" [23]. This potent idea has been renamed soon after as "multivariant execution".

There is a wide range of realizations of MVE in different contexts. Bruschi et al. [16] and Salamat et al. [50] pioneered the idea of executing the variants in parallel. Subsequent techniques focus on MVE for mitigating memory vulnerabilities [31, 41] and other specific security problems including return-oriented programming attacks [58] and code injection [52]. A key design decision of MVE is whether it is achieved in kernel space [47], in user-space [51], with exploiting hardware features [36], or even throught code polymorphism [10]. Finally, one can neatly exploit the limit case of executing only two variants [35, 43]. The body of research on MVE in a distributed setting has been less researched. Notably, Voulimeneas et al. proposed a multivariant execution system by parallelizing the execution of the variants in different machines [59] for sake of efficiency.

In this paper, we propose an original kind of MVE in the context of edge computing. We generate multiple program variants, which we execute on edge computing nodes. We use the natural redundancy of Edge-Cloud computing architectures to deploy an internet-based MVE. Next section goes into the details of our procedure to generate variants and assemble them into multivariant binaries.

3 MEWE: MULTIVARIANT EXECUTION FOR EDGE COMPUTING

In this section we present MEWE, a novel technique to synthesize multivariant binaries and deploy them on an edge computing platform.

3.1 Overview

The goal of MEWE is to synthesize multivariant WebAssembly binaries, according to the threat model presented in Section 3.2.1. The tool generates application-level multivariant binaries, without any change to the operating system or WebAssembly runtime. The core idea of MEWE is to synthesize diversified function variants providing execution-path randomization, according to the diversity model presented in Section 3.2.2.

In Figure 1, we summarize the analysis and transformation pipeline of MEWE. We pass a bitcode to be diversified, as an input to MEWE. Analysis and transformations are performed at the level of LLVM's intermediate representation (LLVM IR), as it is the best format for us to perform our modifications (see Section 3.2.3). LLVM binaries can be obtained from any language with an LLVM frontend such as C/C++, Rust or Go, and they can easily be compiled to WebAssembly. In Step ①, the binary is passed to CROW [18], which is a superdiversifier for Wasm that generates a set of variants for the functions in the binary. Step ② packages all the variants in one single multivariant LLVM binary. In Step ③, we use a special component, called a "mixer", which augments the binary with two different components: an HTTP endpoint harness and a random generator, which are both required for executing Wasm at the edge. The harness is used to connect the program to its execution environment while the generator provides support for random execution path at runtime. The final output of Step ④ is a standalone multivariant WebAssembly binary that can be deployed on an edge-cloud computing platform. In the following sections, we describe in greater details the different stages of the workflow.

3.2 Key design choices

In this section, we introduce the main design decisions behind MEWE, starting from the threat model, to aligning the code analysis and transformation techniques.

3.2.1 Threat Model. As we describe in Section 2.2, to benefit from the performance improvements offered by edge computing, developers modularize their services into a set of WebAssembly functions. The binaries are then deployed on all the nodes provided by the edge computing platforms. However, this model of distributing the exact same WebAssembly binary on hundreds of computation nodes is a serious risk for the infrastructure: a malicious developer who manages to exploit one vulnerability in one edge location can exploit all the other locations with the same attack vector.

With MEWE, we aim to defend against an attacker that perform BOBE attacks. These attacks include but are not limited to timing specific operations [6, 11], counting register spill/reload operations to study and exploit memory [48] and performing call stack analysis. They can be performed either locally or remotely by finding a vulnerability or using shared resources in the case of a multi-tenant Edge computing server but the details of such exploitation are out of scope of this study.

3.2.2 Execution Diversification Model. MEWE is designed to randomize the execution of WebAssembly programs, via diversification transformations. Per Crane et al. those transformations are made to hinder side-channel attacks [24]. All programs are diversified with behavior preservation guarantees [18]. The core diversification strategies are: (1) *Constant Inferring.* MEWE identifies variables whose value can computed at compile time and are used to control branches. This has an effect on program execution times [15]. (2) *Call Stack Randomization.* MEWE introduces equivalent synthetic functions that are called randomly. This results in randomized call stacks, which complicates attacks based on call stack analysis [40]. (3) *Inline Expansion.* MEWE inlines methods when appropriate. This also results in different call stacks, to hinder the same kind of attacks as for call stack randomization [40]. (4) *Spills/Reloads.* By performing semantically equivalent transformations for arithmetic expressions, the number of register spill/reload operations changes. Therefore, this changes the memory accesses in the machine code that is executed, affecting the measurement of memory side-channels [48].

3.2.3 Diversification at the LLVM level. MEWE diversifies programs at the LLVM level. Other solutions would have been to diversify at the source code level [8], or at the native binary level, eg x86 [22]. However, the former would limit the applicability of our work. The latter is not compatible with edge computing: the top edge computing execution platforms, e.g. Cloudflare and Fastly, mostly take WebAssembly binaries as input.

LLVM, on the contrary, does not suffer from those limitations: 1) it supports different languages, with a rich ecosystem of frontends 2) it can reliably be retargeted to WebAssembly, thanks to the corresponding mature component in the LLVM toolchain.

3.3 Variant generation

MEWE relies on the superdiversifier CROW [18] to automatically diversify each function in the input LLVM binary (Step ①). CROW receives an LLVM module, analyzes the binary at the function block level and generates semantically equivalent variants for each function, if they exist. A function variant for MEWE is semantically equivalent to the original (i.e., same input/output behavior), but exhibits a different internal behavior through tracing. Since the variants created by CROW are artificially synthesized from the original binary, after Step ①, they are necessarily equivalent to the original program.

3.4 Combining variants into multivariant functions

Step ② of MEWE consists in combining the variants generated for the original functions, into into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function for which we generate variants. A dispatcher function is a synthetic function that is in charge of choosing a variant at random, every time the original function is invoked during the execution. The random invocation of different variants at runtime is a known randomization technique, for example used by Lettner et al. with sanitizers [39].

With the introduction of dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

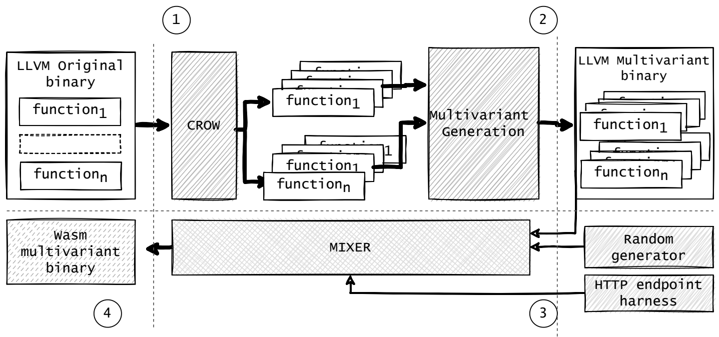


Fig. 1. Overview of MEWE. It takes as input the LLVM binary representation of a service composed of multiple functions. It first generates a set of functionally equivalent variants for each function in the binary and then generates a LLVM multivariant binary composed of all the function variants as well as dispatcher functions in charge of selecting a variant when a function is invoked. The MEWE mixer composes the LLVM multivariant binary with a random number generation library and an edge specific HTTP harness, in order to produce a WebAssembly multivariant binary accessible through an HTTP endpoint and ready to be deployed to the edge.

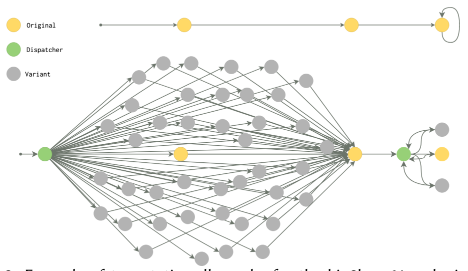


Fig. 2. Example of two static call graphs for the bin2base64 endpoint of libsodium. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph $\langle N, E \rangle$ where the nodes in N represent all the functions in the binary and an edge $(f_1, f_2) \in E$ represents a possible invocation of f_2 by f_1 [49], where the nodes are typed. The nodes in N have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

In Figure 2, we show the original static call graph for program bin2base64 (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function dispatchers and the yellow nodes are the original functions. The possible calls are represented by the directed edges. The original bin2base64 includes 3 functions. MEWE generates 43 variants for the first function, none for the second and three for the third function. MEWE introduces two dispatcher nodes, for the first and third functions. Each dispatcher is connected to the corresponding function variants, in order to invoke one variant randomly at runtime.

The right most green node of Figure 2 is a function constructed as follows (See code in Appendix A). The function body first calls the random generator, which returns a value that is then used to invoke a specific function variant. It should be noted that the dispatcher function is constructed using the same signature as the original function.

We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution based attacks [45]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [33]. This also allows MEWE to inline function variants inside the dispatcher, instead of defining them again. Here we trade security over performance, since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

3.5 MEWE's Mixer

The MEWE mixer has four specific objectives: wrap functions as HTTP endpoints, link the LLVM multivariant binary, inject a random generator and merge all these components into a multivariant WebAssembly binary.

We use the Rustc compiler[1] to orchestrate the mixing. For the generator, we rely on WASI's specification [5] for the random behavior of the dispatchers. Its exact implementation is dependent on the platform on which the binary is deployed. For the HTTP harnesses, since our edge computing use case is based on the Fastly infrastructure, we rely on the Fastly API[2] to transform our Wasm binaries into HTTP endpoints. The harness enables a function to be called as an HTTP request and to return a HTTP response. Throughout this paper, we refer to an endpoint as the closure of invoked functions when the entry point of the WebAssembly binary is executed.

3.6 Implementation

The multivariant combination (Step ②) is implemented in 942 lines of C++ code. Its uses the LLVM 12.0.0 libraries to extend the LLVM standard linker tool capability with the multivariant generation.

[1]https://doc.rust-lang.org/rustc/what-is-rustc.html
[2]https://docs.rs/crate/fastly/0.7.3

MEWE's Mixer (Step ③) is implemented as an orchestration of the rustc and the WebAssembly backend provided by CROW. An instantiation of how the multivariant binary works can be appreciated at Appendix B. For sake of open science and for fostering research on this important topic, the code of MEWE is made publicly available on GitHub: https://github.com/Jacarte/MEWE.

4 EXPERIMENTAL METHODOLOGY

In this section we introduce our methodology to evaluate MEWE. First, we present our research questions and the services with which we experiment the generation and the execution of multivariant binaries. Then, we detail the methodology for each research question.

4.1 Research questions

To evaluate the capabilities of MEWE, we formulate the following research questions:

RQ1: **(Multivariant Generation) How much diversity can MEWE synthesize and embed in a multivariant binary ?** MEWE packages function variants in multivariant binaries. With this first question, we aim at measuring the amount of diversity that MEWE can synthesize in the call graph of a program.

RQ2: **(Intra MVE) To what extent does MEWE achieve multivariant executions on an edge compute node?** With this question we assess the ability of MEWE to produce binaries that actually exhibit random execution paths when executed on one edge node.

RQ3: **(Internet MVE) To what extent does MEWE achieve multivariant execution over the worldwide Fastly infrastructure?** We check the diversity of execution traces gathered from the execution of a multivariant binary. The traces are collected from all edge nodes in order to assess MVE at a worldwide scale.

RQ4: **What is the impact of the proposed multi-version execution on timing side-channels?** MEWE generates binaries that embed a multivariant behavior. We measure to what extent MEWE generates different execution times on the edge. Then, we discuss how multivariant binaries contribute to less predictable timing side-channels.

The core of the validation methodology for our tool MEWE, consists in building multivariant binaries for several, relevant endpoints and to deploy and execute them on the Fastly edge-cloud platform.

4.2 Study subjects

We select two mature and typical edge-cloud computing projects to study the feasibility of MEWE. The projects are selected based on: suitability for diversity synthesis with CROW (the projects should have the ability to collect their modules in LLVM intermediate representation), suitability for deployment on the Fastly infrastructure (the project should be easily portable Wasm/WASI and compatible with the Rust Fastly API), low chances to hit execution paths with no dispatchers and possibility to collect their execution runtime information (the endpoints should execute in a reasonable time of maximum 1 second even with the overhead of instrumentation). The selected projects are: **libsodium**, an encryption, decryption, signature and password hashing library which can be ported to WebAssembly and **qrcode-rust**, a QrCode and MicroQrCode generator written in Rust.

Name	#Endpoints	#Functions	#Instr.
libsodium https://github.com/ jedisct1/libsodium	5	62	6187
qrcode-rust https://github.com/ kennytm/qrcode-rust	2	1840	127700

Table 1. Selected projects to evaluate MEWE: project name; the number of endpoints in the project that we consider for our experiments, the total number of functions to implement the endpoints, and the total number of WebAssembly instructions in the original binaries.

In Table 1, we summarize some key metrics that capture the relevance of the selected projects. The table shows the project name with its repository address, the number of selected endpoints for which we build multivariant binaries, the total number of functions included in the endpoints and the total number of Wasm instructions in the original binary. Notice that, the metadata is extracted from the Wasm binaries before they are sent to the edge-cloud computing platform, thus, the number of functions might be not the same in the static analysis of the project source code

4.3 Experiment's platform

We run all our experiments on the Fastly edge computing platform. We deploy and execute the original and the multivariant endpoints on 64 edge nodes located around the world[3]. These edge nodes usually have an arbitrary and heterogeneous composition in terms of architecture and CPU model. The deployment procedure is the same as the one described in Section 2.2. The developers implement and compile their services to WebAssembly. In the case of Fastly, the WebAssembly binaries need to be implemented with the Fastly platform API specification so they can properly deal with HTTP requests. When the compiled binary is transmitted to Fastly, it is translated to x86 machine code with Lucet, which ensures the isolation of the service.

4.4 RQ1 Multivariant diversity

We run MEWE on each endpoint function of our 7 endpoints. In this experiment, we bound the search for function variant with timeout of 5 minutes per function. This produces one multivariant binary for each endpoint. To answer RQ1, we measure the number of function variants embedded in each multivariant binary, as well as the number of execution paths that are added in the mutivariant call graphs, thanks to the function variants.

4.5 RQ2 Intra MTD

We deploy the multivariant binaries of each of the 7 endpoints presented in Table 2, on the 64 edge nodes of Fastly. We execute each endpoint, multiple times on each node, to measure the diversity of execution traces that are exhibited by the multivariant binaries. We have a time budget of 48 hours for this experiment. Within this

[3]The number of nodes provided in the whole platform is 72, we decided to keep only the 64 nodes that remained stable during our experimentation.

timeframe, we can query each endpoint 100 times on each node. Each query on the same endpoint is performed with the same input value. This is to guarantee that, if we observe different traces for different executions, it is due to the presence of multiple function variants. The input values are available as part of our reproduction package.

For each query, we collect the execution trace , i.e., the sequence of function names that have been executed when triggering the query. To observe these traces, we instrument the multivariant binaries to record each function entrance.

To answer RQ2, we measure the number of unique execution traces exhibited by each multivariant binary, on each separate edge node. To compare the traces, we hash them with the `sha256` function. We then calculate the number of unique hashes among the 100 traces collected for an endpoint on one edge node. We formulate the following definitions to construct the metric for RQ3.

Metric 1. *Unique traces: $R(n,e)$. Let $S(n,e) = \{T_1, T_2, ..., T_{100}\}$ be the collection of 100 traces collected for one endpoint e on an edge node n, $H(n,e)$ the collection of hashes of each trace and $U(n,e)$ the set of unique trace hashes in $H(n,e)$. The uniqueness ratio of traces collected for edge node n and endpoint e is defined as*

$$R(n,e) = \frac{|U(n,e)|}{|H(n,e)|}$$

The inputs that we pass to execute the endpoints at the edge and the received output for all executions are available in the reproduction repository at https://github.com/Jacarte/MEWE.

4.6 RQ3 Inter MTD

We answer RQ3 by calculating the normalized Shannon entropy for all collected execution traces for each endpoint. We define the following metric.

Metric 2. *Normalized Shannon entropy: $E(e)$ Let e be an endpoint, $C(e) = ;_{n=0}^{64} H(n,e)$ be the union of all trace hashes for all edge nodes. The normalized Shannon Entropy for the endpoint e over the collected traces is defined as:*

$$E(e) = -\Sigma \frac{p_x * log(p_x)}{log(|C(e)|)}$$

Where p_x is the discrete probability of the occurrence of the hash x over $C(e)$.

Notice that we normalize the standard definition of the Shannon Entropy by using the perfect case where all trace hashes are different. This normalization allows us to compare the calculated entropy between endpoints. The value of the metric can go from 0 to 1. The worst entropy, value 0, means that the endpoint always perform the same path independently of the edge node and the number of times the trace is collected for the same node. On the contrary, 1 for the best entropy, when each edge node executes a different path every time the endpoint is requested.

The Shannon Entropy gives the uncertainty in the outcome of a sampling process. If a specific trace has a high frequency of appearing in part of the sampling, then it is certain that this trace will appear in the other part of the sampling.

We calculate the metric for the 7 endpoints, for 100 traces collected from 64 edge nodes, for a total of 6400 collected traces per endpoint. Each trace is collected in a round robin strategy, i.e., the traces are collected from the 64 edge nodes sequentially. For example, we collect the first trace from all nodes before continuing to the collection of the second trace. This process is followed until 100 traces are collected from all edge nodes.

4.7 RQ4 Timing side-channels

For each endpoint listed in Table 2, we measure the impact of MEWE on timing. For this, we use the following metric:

Metric 3. *Execution time: For a deployed binary on the edge, the execution time is the time spent on the edge to execute the binary.*

Note that edge-computing platforms are, by definition, reached from the Internet. Consequently, there may be latency in the timing measurement due to round-trip HTTP requests. This can bias the distribution of measured execution times for the multivariant binary. To avoid this bias, we instrument the code to only measure the execution on the edge nodes.

We collect 100k execution times for each binary, both the original and multivariant binaries. We perform a Mann-Withney U test [42] to compare both execution time distributions. If the P-value is lower than 0.05, two compared distributions are different.

5 EXPERIMENTAL RESULTS

5.1 RQ1 Results: Multivariant generation

We use MEWE to generate a multivariant binary for each of the 7 endpoints included in our 2 study subjects. We then calculate the number of diversified functions, in each endpoint, as well as how they combine to increase the number of possible execution paths in the static call graph for the original and the multivariant binaries.

The sections 'Original binary' and 'Multivariant WebAssembly binary' of Table 2 summarize the key data for RQ1. In the 'Original binary' section, the first column (#F) gives the number of functions in the original binary and the second column (#Paths) gives the number of possible execution paths in the original static call graph. The 'Multivariant WebAssembly binary' section first shows the number of each type of nodes in the multivariant call graph: #Non div. is the number of original functions that could not be diversified by MEWE, #D is the number of dispatcher nodes generated by MEWE for each function that was successfully diversified, and #V is the total number of function variants generated by MEWE. The last column of this section is the number of possible execution paths in the static multivariant call graph.

For all 7 endpoints, MEWE was able to diversify several functions and to combine them in order to increase the number of possible execution paths in several orders of magnitude. For example, in the case of the `encrypt` function of libsodium, the original binary contains 23 functions that can be combined in 4 different paths. MEWE generated a total of 56 variants for 5 of the 23 functions. These variants, combined with the 18 original functions in the multivariant call graph, form 325 execution paths. In other words, the number of possible ways to achieve the same encryption function has increased from 4 to 325, including dispatcher nodes that are in charge of randomizing the choice of variants at 5 different locations of the call graph. This increased number of possible paths, combined with random choices, made at runtime, increases the effort a potential attacker needs to guess what variant is executed and hence what vulnerability she can exploit.

	Original binary		Multivariant WebAssembly binary			
Endpoint	#F	#Paths	#Non D	#D	#V	#Paths
libsodium						
encrypt	23	4	18	5	56	325
decrypt	20	3	16	5	49	84
random	8	2	6	2	238	12864
invert	8	2	6	2	125	2784
bin2base64	3	2	1	2	47	172
qrcode-rust						
qr_str	982	$688*10^6$	965	17	2092	$97*10^{12}$
qr_image	858	$1.4*10^6$	843	15	2063	$3*10^9$

Table 2. Static diversity generated by MEWE, measured on the static call graphs of the WebAssembly binaries, and the preservation of this diversity after translation to machine code. The table is structured as follows: Endpoint name; number of functions and numbers of possible paths in the original WebAssembly binary call graph; number of non diversified functions, number of created dispatchers (one per diversified functions), total number of function variants and number of execution paths in the multivariant WebAssembly binary call graph.

We have observed that there is no linear correlation between the number of diversified functions, the number of generated variants and the number of execution paths. We have manually analyzed the endpoint with the largest number of possible execution paths in the multivariant Wasm binary: `qr_str` of qrcode-rust. MEWE generated 2092 function variants for this endpoint. Moreover, MEWE inserted 17 dispatchers in the call graph of the endpoint. For each dispatcher, MEWE includes between 428 and 3 variants. If the original execution path contains function for which MEWE is able to generate variants, then, there is a combinatorial explosion in the number of execution paths for the generated Wasm multivariant module. The increase of the possible execution paths theoretically augments the uncertainty on which one to perform, in the latter case, approx. 140 000 times. As Cabrera and colleagues observed [18] for CROW, a large presence of loops and arithmetic operations in the original function code leverages to more diversification.

Looking at the #D (#Dispatchers) and #V (#Variants) columns of the 'Multivariant WebAssembly binary' section of Table 2, we notice that the number of variants generated per function greatly varies. For example, for both the `invert` and the `bin2base64` functions of Libsodium, MEWE manages to diversify 2 functions (reflected by the presence of 2 dispatcher nodes in the multivariant call graph). Yet, MEWE generates a total of 125 variants for the 2 functions in `invert`, and only 47 variants for the 2 functions in `bin2base64`. The main reason for this is related to the complexity of the diversified functions, which impacts the opportunities for the synthesis of code variations.

Columns #Non D of the 'Multivariant WebAssembly binary' section of Table 2 indicates that, in each endpoint, there exists a number of functions for which MEWE did not manage to generate variants. We identify three reasons for this, related to the diversification procedure of CROW, used by MEWE to diversify individual functions. First, some functions cannot be diversified by CROW, e.g., functions that wrap only memory operations, which are oblivious to CROW diversification technique. Second, the complexity of the function directly affects the number of variants that CROW can generate. Third, the diversification procedure of CROW is essentially a search procedure, which results are directly impacted by the tie budget for the search. In all experiments we give CROW 5 minutes maximum to synthesize function variants, which is a low budget for many functions. It is important to notice that, the successful diversification of some functions in each endpoint, and their combination within the call graph of the endpoint, dramatically increases the number of possible paths that can triggered for multivariant executions.

Answer to RQ1: MEWE dramatically increases the number of possible execution paths in the multivariant WebAssembly binary of each endpoint. The large number of possible execution paths, combined with multiple points of random choice in the multivariant call graph thwart the prediction of which path will be taken at runtime.

5.2 RQ2 Results: Intra MTD

To answer RQ2, we execute the multivariant binaries of each endpoint, on the Fastly edge-cloud infrastructure. We execute each endpoint 100 times on each of the 64 Fastly edge nodes. All the executions of a given endpoint are performed with the same input. This allows us to determine if the execution traces are different due to the injected dispatchers and their random behavior. After each execution of an endpoint, we collect the sequence of invoked functions, i.e., the execution trace. Our intuition is that the random dispatchers combined with the function variants embedded in a multivariant binary are very likely to trigger different traces for the same execution, i.e., when an endpoint is executed several times in a row with the same input and on the same edge node. The way both the function variants and the dispatchers contribute to exhibiting different execution traces is illustrated in Figure 6.

Figure 3 shows the ratio of unique traces exhibited by each endpoint, on each of the 64 separate edge nodes. The X corresponds to the edge nodes. The Y axis gives the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point in the Z axis representing Metric 1 over 100 execution traces.

For all edge nodes, the ratio of unique traces is above 0.38. In 6 out of 7 cases, we have observed that the ratio is remarkably high, above 0.9. These results show that MEWE generates multivariant binaries that can randomize execution paths at runtime, in the context of an edge node. The randomization dispatchers, associated to a significant number of function variants greatly reduce the certainty about which computation is performed when running a specific input with a given input value.

Let's illustrate the phenomenon with the endpoint `invert`. The endpoint `invert` receives a vector of integers and returns its inversion. Passing a vector of integers with 100 elements as input, $I = [100, ..., 0]$, results in output $O = [0, ..., 100]$. When the endpoint executes 100 times with the same input on the original binary, we observe 100 times the same execution trace. When the endpoint is executed 100 times with the same input I on the multivariant binary, we observe between 95 and 100 unique execution traces, depending on the edge node. Analyzing the traces we observe that they include only two invocations to a dispatcher, one at the start of the trace and one at the end. The remaining events in the trace are fixed each time the endpoint is executed with the same input I. Thus, the maximum number of possible unique traces is the multiplication of the number of variants for each dispatcher, in this case $29 \times 96 = 2784$. The probability of observing the same trace is $1/2784$.

For multivariant binaries that embed only a few variants, like in the case of the `bin2base64` endpoint, the ratio of unique traces per node is lower than for the other endpoints. With the input we pass to

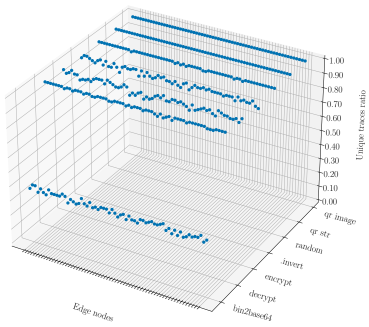


Fig. 3. Ratio of unique execution traces for each endpoint on each edge node. The X axis illustrates the edge nodes. The Y axis annotates the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point representing the Metric 1 value in a set of 100 collected execution traces.

`bin2base64`, the execution trace includes 57 function calls. We have observed that, only one of these calls invokes a dispatcher, which can select among 41 variants. Thus, probability of having the same execution trace twice is 1/41.

Meanwhile, `qr_str` embeds thousands of variants, and the input we pass triggers the invocation of 3M functions, for which 210666 random choices are taken relying on 17 dispatchers. Consequently, the probability of observing the same trace twice is infinitesimal. Indeed, all the executions of `qr_str` are unique, on each separate edge node. This is shown in Figure 3, where the ratio of unique traces is 1 on all edge nodes.

Answer to RQ2: Repeated executions of a multivariant binary with the same input on an individual edge node exhibits diverse execution traces. MEWE successfully synthesizes multivariant binaries that trigger diverse execution paths at runtime, on individual edge nodes.

5.3 RQ3 Results: Internet MTD

To answer RQ3, we build the union of all the execution traces collected on all edge nodes for a given endpoint. Then, we compute the normalized Shannon Entropy over this set for each endpoint (Metric 2). Our goal is to determine whether the diversity of execution traces we observed on individual nodes in RQ3, actually generalizes to the whole edge-cloud infrastructure. Depending on many factors, such as the random number generator or a bug in the dispatcher, it could happen that we observe different traces on individual nodes, but that the set of traces is the same on all nodes. With RQ4 we assess the ability of MEWE to exhibit multivariant execution at a global scale.

Table 3 provides the data to answer RQ3. The second column gives the normalized Shannon Entropy value (Metric 2). Columns 3 and 4 give the median and the standard deviation for the length of the execution traces. Columns 5 and 6 give the number of dispatchers that are invoked during the execution of the endpoint (#ED) and the total number of invocations of these endpoints (#Rch). These last two columns indicate to what extent the execution paths are actually randomized at runtime. In the cases of `invert` and `random`, both have the same

Endpoint	Entropy	MTL	σ	#ED	#RCh
libsodium					
encrypt	0.87	816	0	5	4M
decrypt	0.96	440	0	5	2M
random	0.98	15	5	2	12800
invert	0.87	7343	0	2	12800
bin2base64	0.42	57	0	1	6400
qrcode-rust					
qr_str	1.00	3045193	0	17	1348M
qr_image	1.00	3015450	0	15	1345M

Table 3. Execution trace diversity over the Fastly edge-cloud computing platform. The table is formed of 6 columns: the name of the endpoint, the normalized Shannon Entropy value (Metric 2), the median size of the execution traces (MTL), the standard deviation for the trace lengths the number of executed dispatchers (#ED) and the number of total random choices taken during all the 6400 executions (#RCh).

number of taken random choices. However, the number of variants to chose in `random` are larger, thus, the entropy, is larger than `invert`.

Overall, the normalized Shannon Entropy is above 42%. This is evidence that the multivariant binaries generated by MEWE can indeed exhibit a high degree of execution trace diversity, while keeping the same functionality. The number of randomization points along the execution paths (#Rch) is at the core of these high entropy values. For example, every execution of the `encrypt` endpoint triggers 4M random choices among the different function variants embedded in the multivariant binaries. Such a high degree of randomization is essential to generate very diverse execution traces.

The `bin2base64` endpoint has the lowest level of diversity. As discussed in RQ2, this endpoint is the one that has the least variants and its execution path can be randomized only at one point. The low level of unique traces observed on individual nodes is reflected at the system wide scale with a globally low entropy.

For both `qr_str` and `qr_image` the entropy value is 1.0. This means that all the traces that we observe for all the executions of these endpoints are different from each other. In other words, someone who runs these services over and over with the same input cannot know exactly what code will be executed in the next execution. These very high entropy values are made possible by the millions of random choices that are made along the execution paths of these endpoints.

While there is a high degree of diversity among the traces exhibited by each endpoint, they all have the same length, except in the case of `random`. This means that the entropy is a direct consequence of the invocations of the dispatchers. In the case of `random`, it naturally has a non-deterministic behavior. Meanwhile, we observe several calls to dispatchers in during the execution of the multivariant binary, which indicates that MEWE can amplify the natural diversity of traces exhibited by `random`. For each endpoint, we managed to trigger all dispatchers during its execution. There is a correlation between the entropy and the number of random choices (Column #RChs) taken during the execution of the endpoints. For a high number of dispatchers, and therefore random choices, the entropy is large, like the cases of `qr_str` and `qr_image` show. The contrary happens to `bin2base64` where its multivariant binary contains only one dispatcher.

	Original bin.		**Multivariant Wasm**	
Endpoint	Median (μs)	σ	Median (μs)	σ
libsodium				
encrypt	7	5	217	43
decrypt	13	6	225	47
random	16	7	232	53
invert	119	34	341	65
bin2base64	10	5	215	35
qrcode-rust				
qr_str	3,117	418	492,606	36,864
qr_image	3,091	412	512,669	41,718

Table 4. Execution time distributions for 100k executions, for the original endpoints and their corresponding multivariants. The table is structured in two sections. The first section shows the endpoint name, the median execution time and its standard deviation for the original endpoint. The second section shows the median execution time and its standard deviation for the multivariant WebAssembly binary.

Answer to RQ3: At the internet scale of the Edge platform, the multivariant binaries synthesized by MEWE exhibit a massive diversity of execution traces, while still providing the original service. It is virtually impossible for an attacker to predict which is taken for a given query.

5.4 RQ4 Results: Timing side-channels

For each endpoint used in RQ1, we compare the execution time distributions for the original binary and the multivariant binary. All distributions are measured on 100k executions. In Table 4, we show the execution time for the original endpoints and their corresponding multivariant. The table is structured in two sections. The first section shows the endpoint name, the median and standard deviation of the original endpoint. The second section shows the median and the standard deviation for the execution time of the corresponding multivariant binary.

We also observe that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 4, each subplot represents the distribution for a single endpoint, with the colors blue and green representing the original and multivariant binary respectively. These plots reveal that the execution times are indeed spread over a larger range of values compared to the original binary. This is evidence that execution time is less predictable for multivariant binaries than for the original ones.

We evaluate to what extent a specific variant can be detected by observing the execution time distribution. This evaluation is based on the measurement with one endpoint. For this, we choose endpoint `bin2base64` because it is the end point that has the least variants and the least dispatchers, which is the most conservative assumption.

We dissect the collected execution times for the `bin2base64` endpoint, grouping them by execution path. In Figure 5, each opaque curve represents a cumulative execution time distribution of a unique execution path out of the 41 observed. We observe that no specific distribution is remarkably different from another one. Thus, no specific variant can be inferred out of the projection of all execution times like the ones presented in Figure 4. Nevertheless, we calculate a Mann-Whitney test for each pair of distributions, 41×41 pairs. For all cases, there is no statistical evidence that the distributions are different, $P > 0.05$.

Recall that the choice of function variant is randomized at each function invocation, and the variants have different execution times as a consequence of the code transformations, i.e., some variants execute more instructions than others. Consequently, attacks relying on measuring precise execution times of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

We evaluate the impact of multivariant binaries on execution time. As a baseline, we consider the evaluation proposed by Fastly [1, 2]: a Markdown to HTML conversion service shall run on their edge platform and return a response in less than 100 ms, allowing one request for every single keystroke. In this context, all the multivariant binaries for Libsodium match the baseline and still support requests at the speed of keystrokes. The multivariant binaries for QR encoding respond in a reasonable time for end users, i.e., in less than half a second, but are below the baseline. In general, we note that the execution times are slower for multivariant binaries. Being under 500 ms in general, this does not represent a threat to the applicability of multivariant execution at the edge. Yet, it calls for future optimization research.

Answer to RQ4: The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. MEWE contributes to mitigate potential attacks based on predictable execution times.

6 RELATED WORK

Our work is in the area of software diversification for security, a research field discovered by researchers Forrest [26] and Cohen [21]. We contribute a novel technique for multivariant execution, and discuss related work in Section 2. Here, we position our contribution with respect to previous work on randomization and security for WebAssembly.

6.1 Related Work on Randomization

A randomization technique creates a set of unique executions for the very same program [12]. Seminal works include instruction-set randomization [9, 34] to create a unique mapping between artificial CPU instructions and real ones. This makes it very hard for an attacker ignoring the key to inject executable code. Compiler-based techniques can randomly introduce NOP and padding to statically diversify programs. [30] have explored how to use NOP and it breaks the predictability of program execution, even mitigating certain exploits to an extent.

Chew and Song [19] target operating system randomization. They randomize the interface between the operating system and the user applications: the system call numbers, the library entry points (memory addresses) and the stack placement. All those techniques are dynamic, done at runtime using load-time preprocessing and rewriting. Bathkar et al. [12, 13] have proposed three kinds of randomization transformations: randomizing the base addresses of applications and libraries memory regions, random permutation of the order of variables and

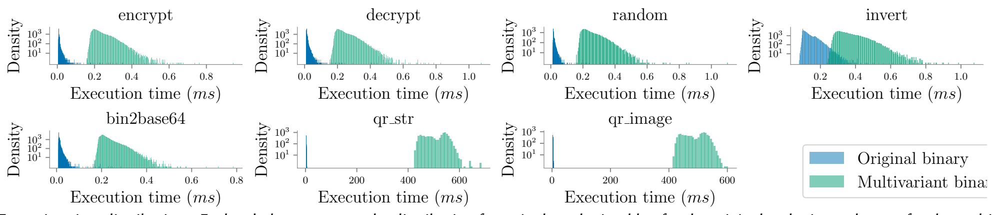


Fig. 4. Execution time distributions. Each subplot represents the distribution for a single endpoint, blue for the original endpoint and green for the multivariant binary. The X axis shows the execution time in milliseconds and the Y axis shows the density distribution in logarithmic scale.

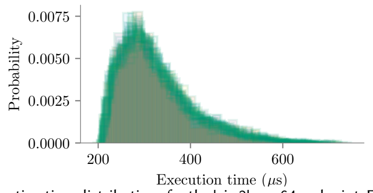


Fig. 5. Execution time distributions for the `bin2base64` endpoint. Each opaque curve represents an execution time distribution of a unique execution path out of the 41 observed.

routines, and the random introduction of random gaps between objects. Dynamic randomization can address different kinds of problems. In particular, it mitigates a large range of memory error exploits. Recent work in this field include stack layout randomization against data-oriented programming [7] and memory safety violations [37], as well as a technique to reduce the exposure time of persistent memory objects to increase the frequency of address randomization [60].

We contribute to the field of randomization, at two stages. First, we automatically generate variants of a given program, which have different WebAssembly code and still behave the same. Second, we randomly select which variant is executed at runtime, creating a multivariant execution scheme that randomizes the observable execution trace at each run of the program.

Davi et al. proposed Isomeron [25], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. With this strategy, a potential attacker cannot predict whether the original or the variant of a program will execute. MEWE proposes two key novel contributions. First, our code diversification step can generate variants of complex control flow structures by inferring constants or loop unrolling. Second, MEWE interconnects hundreds of variants and several randomization dispatchers in a single binary, increasing by orders of magnitude the runtime uncertainty about what code will actually run, compared to the choice among 2 variants proposed by Isomeron.

6.2 Related work on WebAssembly Security

The reference piece about WebAssembly security is by Lehmann et al. [38], which presents three attack primitives. Lehmann et al. have then followed up with a large-scale empirical study of WebAssembly binaries [29]. Narayan et al. [45] remark that the security model of WebAssembly is vulnerable to Spectre attacks. This means that WebAssembly sandboxes may be hijacked and leak memory. They propose to modify the Lucet compiler used by Fastly to incorporate LLVM fence instructions[4] in the machine code generation, trying to avoid speculative execution mistakes. Johnson et al. [33], on the other hand, propose fault isolation for WebAssembly binaries, a technique that can be applied before being deployed to the edge-cloud platforms. Stievenart et al. [55] design a static analysis dedicated to information flow problems. Bian et al. [14] performs runtime monitoring of WebAssembly to detect cryptojacking. The main difference with our work is that our defense mechanism is larger in scope, meant to tackle "yet unknown" vulnerabilities. Notably, MEWE is agnostic from the last-step compilation pass that translates Wasm to machine code, which means that the multivariant binaries can be deployed on any edge-cloud platform that can receive WebAssembly endpoints, regardless of the underlying hardware.

7 CONCLUSION

In this work we propose a novel technique to automatically synthesize multivariant binaries to be deployed on edge computing platforms. Our tool, MEWE, operates on a single service implemented as a WebAssembly binary. It automatically generates functionally equivalent variants for each function that implements the service, and combines all the variants in a single WebAssembly binary, which exact execution path is randomized at runtime. Our evaluation with 7 real-world cryptography and QR encoding services shows that MEWE can generate hundreds of function variants and combine them into binaries that include from thousands to millions of possible execution paths. The deployment and execution of the multivariant binaries on the Fastly cloud platform showed that they actually exhibit a very high diversity of execution at runtime, in single edge nodes, as well as Internet scale.

Future work with MEWE will address the trade-off between a large space for execution path randomization and the computation cost of large-scale runtime randomization. In addition, the synthesis of a large pool of variants supports the exploration of the concurrent execution of multiple variants to detect misbehaviors in services deployed at the edge. Besides, several components of MEWE are implemented to operate at the level of the LLVM intermediate language. These components are compatible with other LLVM workflows. We plan to extend MEWE for other LLVM workflows, such as Rust, a popular workflow for Wasm applications and libraries.

REFERENCES

[1] 2020. Markdown to HTML. https://markdown-converter.edgecompute.app/

[4] https://llvm.org/doxygen/classllvm_1_1FenceInst.html

[2] 2020. The power of serverless, 72 times over. https://www.fastly.com/blog/the-power-of-serverless-at-the-edge

[3] 2021. Global CDN Disruption. https://status.fastly.com/incidents/vpk0ssybt3bj

[4] 2021. The New York Times on failure, risk, and prepping for the 2016 US presidential election – Fastly. https://www.fastly.com/blog/new-york-times-on-failure-risk-and-prepping-2016-us-presidential-election

[5] 2021. WebAssembly System Interface. https://github.com/WebAssembly/WASI

[6] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. 2007. Cache based remote timing attack on the AES. In *Cryptographers' track at the RSA conference*. Springer, 271–286.

[7] Misiker Tadesse Aga and Todd Austin. 2019. Smokestack: thwarting DOP attacks with runtime stack layout randomization. In *Proc. of CGO*. 26–36. https://drive.google.com/file/d/12TvsrgL8Wt6IMfe6ASUp8y69L-bCVao0/view

[8] Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, and Maxime Tricoire. 2015. Multitier diversification in Web-based software applications. *IEEE Software* 32, 1 (2015), 83–90. https://doi.org/10.1109/MS.2014.150

[9] Elena Gabriela Barrantes, David H Ackley, Stephanie Forrest, Trek S Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. CCS*. 281–289.

[10] Nicolas Belleville, Damien Couroussé, Karine Heydemann, and Henri-Pierre Charles. 2018. Automated Software Protection for the Masses Against Side-Channel Attacks. *ACM Trans. Archit. Code Optim.* 15, 4, Article 47 (nov 2018), 27 pages. https://doi.org/10.1145/3281662

[11] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).

[12] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium*.

[13] Sandeep Bhatkar, Ron Sekar, and Daniel C DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*. 271–286.

[14] Weikang Bian, Wei Meng, and Mingxue Zhang. 2020. Minethrottle: Defending against wasm in-browser cryptojacking. In *Proceedings of The Web Conference 2020*. 3112–3118.

[15] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: inducing timing side channels through just-in-time compilation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1207–1222.

[16] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. 2007. Diversified process replicæ for defeating memory error exploits. In *Proc. of the Int. Performance, Computing, and Communications Conference*.

[17] David Bryant. 2020. Webassembly outside the browser: A new foundation for pervasive computing. In *Proc. of ICWE 2020*. 9–12.

[18] Javier Cabrera-Arteaga, Orestis Floros Malivitsis, Oscar Vera-Pérez, Benoit Baudry, and Martin Monperrus. 2021. CROW: Code Diversification for WebAssembly. In *MADWeb, NDSS 2021*.

[19] Monica Chew and Dawn Song. 2002. *Mitigating buffer overflows by operating system randomization*. Technical Report CS-02-197. Carnegie Mellon University.

[20] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2014. A hybrid edge-cloud architecture for reducing on-demand gaming latency. *Multimedia systems* 20, 5 (2014), 503–519.

[21] Frederick B Cohen. 1993. Operating system protection through program evolution. *Computers & Security* 12, 6 (1993), 565–584.

[22] Bart Coppens, Bjorn De Sutter, and Jonas Maebe. 2013. Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–26.

[23] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-variant systems: a secretless framework for security through diversity. In *Proc. of USENIX Security Symposium* (Vancouver, B.C., Canada) *(USENIX-SS'06)*. http://dl.acm.org/citation.cfm?id=1267336.1267344

[24] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *NDSS*. 8–11.

[25] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *NDSS*.

[26] Stephanie Forrest, Anil Somayaji, and David H Ackley. 1997. Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems*. IEEE, 67–72.

[27] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[28] Pat Hickey. 2018. *Announcing Lucet: Fastly's native WebAssembly compiler and runtime*. Technical Report. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[29] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021*. 2696–2708.

[30] Todd Jackson. 2012. *On the Design, Implications, and Effects of Implementing Software Diversity for Security*. Ph.D. Dissertation. University of California, Irvine.

[31] Todd Jackson, Christian Wimmer, and Michael Franz. 2010. Multi-variant program execution for vulnerability detection and analysis. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*. 1–4.

[32] Martin Jacobsson and Jonas Wåhslén. 2018. Virtual machine execution for wearables based on webassembly. In *EAI International Conference on Body Area Networks*. Springer, Cham, 381–389.

[33] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. SFI safety for native-compiled Wasm. *NDSS. Internet Society* (2021).

[34] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proc. of CCS*. 272–280.

[35] Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. 2015. Dual Execution for On the Fly Fine Grained Execution Comparison. *SIGPLAN Not.* (2015).

[36] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 431–442.

[37] Seongman Lee, Hyeonwoo Kang, Jinsoo Jang, and Brent Byunghoon Kang. 2021. SaVioR: Thwarting Stack-Based Memory Safety Violations by Randomizing Stack Layout. *IEEE Transactions on Dependable and Secure Computing* (2021). https://ieeexplore.ieee.org/iel7/8858/4358699/09369900.pdf

[38] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[39] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. 2018. PartiSan: fast and flexible sanitization via run-time partitioning. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 403–422.

[40] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. 2021. PACStack: an Authenticated Call Stack. In *30th USENIX Security Symposium (USENIX Security 21)*.

[41] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2018. Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing* (2018).

[42] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.* 18, 1 (03 1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[43] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*. 617–630.

[44] P. Mendki. 2020. Evaluating Webassembly Enabled Serverless Approach for Edge Computing. In *2020 IEEE Cloud Summit*. 161–166. https://doi.org/10.1109/IEEECloudSummit48914.2020.00031

[45] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. 2021. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security Symposium*.

[46] Adam J O'Donnell and Harish Sethu. 2004. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and communications security*. 121–131.

[47] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. 2019. kMVX: Detecting kernel information leaks with multi-variant execution. In *ASPLOS*.

[48] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*. 431–446.

[49] Barbara G Ryder. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 3 (1979), 216–226.

[50] Babak Salamat, Andreas Gal, Todd Jackson, Karthik Manivannan, Gregor Wagner, and Michael Franz. 2007. *Stopping Buffer Overflow Attacks at Run-Time: Simultaneous Multi-Variant Program Execution on a Multicore Processor*. Technical Report. Technical Report 07-13, School of Information and Computer Sciences, UCIrvine.

[51] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*. 33–46.

[52] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. 2011. Runtime Defense against Code Injection Attacks Using Replicated Execution. *IEEE Trans. Dependable Secur. Comput.* 8, 4 (2011), 588–601. https://doi.org/10.1109/TDSC.2011.18

[53] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference*. 419–433.

[54] Natalie Silvanovich. 2018. *The Problems and Promise of WebAssembly*. Technical Report. https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html

[55] Quentin Stiévenart and Coen De Roover. 2020. Compositional Information Flow Analysis for WebAssembly Programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 13–24.

[56] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. 2017. On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Comm. Surveys & Tutorials* 19, 3 (2017).

[57] Kenton Varda. 2018. *WebAssembly on Cloudflare Workers*. Technical Report. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/

[58] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. 2015. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing* 13, 4 (2015).

[59] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. 2021. dMVX: Secure and Efficient Multi-Variant Execution in a Distributed Setting. In *Proceedings of the 14th European Workshop on Systems Security*. 41–47.

[60] Yuanchao Xu, Yan Solihin, and Xipeng Shen. 2020. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proc. of ASPLOS*. 987–1000. https://dl.acm.org/doi/pdf/10.1145/3373376.3378492

A DISPATCHER FUNCTION CODE

```
define internal i32 @b64_byte2urlsafe_char(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [ i32 0, label %case_43_ i32 1, label
            %case_44_]
    case_43_: ; preds = %entry
      %2 = call i32 @b64_byte_to_urlsafe_char_43_(%0)
      ret i32 %2
    case_44_: ; preds = %entry
      %3 = <body of b64_byte_to_urlsafe_char_44_>
      ret i32 %3
    end: ; preds = %entry
      %4 = call i32 @b64_byte2urlsafe_char_original(%0)
      ret i32 %4
}
```

Listing 3. Dispatcher function embedded in the multivariant binary of the bin2base64 endpoint of libsodium, which corresponds to the rightmost green node in Figure 2.

B MULTIVARIANT BINARY EXECUTION AT THE EDGE

When a WebAssembly binary is deployed on an edge platform, it is translated to machine code on the fly. For our experiment, we deploy on the production edge nodes of Fastly. This edge computing platform uses Lucet, a native WebAssembly compiler and runtime, to compile and run the deployed Wasm binary [5]. Lucet generates x86 machine code and ensures that the generated machine code executes inside a secure sandbox, controlling memory isolation.

Figure 6 illustrates the runtime behavior of the original and the multivariant binary, when deployed on an Edge node. The top most diagram illustrates the execution trace for the original of the endpoint `bin2base64`. When the HTTP request with the input `"HelloWorld!"` is received, it invokes functions $f1$, $f2$ followed by 27 recursive calls of function $f3$. Then, the endpoint sends the result `"0x000xccv0x10x00b3Jsx130x000x00 0x00xpopAHRvdGE="` of its base64 encoding in an HTTP response.

[5]https://github.com/bytecodealliance/lucet

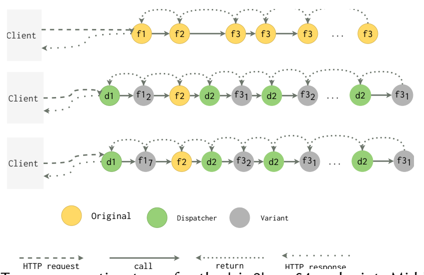


Fig. 6. Top: an execution trace for the `bin2base64` endpoint. Middle and bottom: two different execution traces for the multivariant `bin2base64`, exhibited by two different requests with exactly the same input.

The two diagrams at the bottom of Figure 6 illustrate two executions traces observed through two different requests to the endpoint `bin2base64`. In the first case, the request first triggers the invocation of dispatcher $d1$, which randomly decides to invoke the variant $f1_2$; then $f2$, which has not been diversified by MEWE, is invoked; then the recursive invocations to $f3$ are replaced by iterations over the execution of dispatcher $d2$ followed by a random choice of variants of $f3$. Eventually the result is computed and sent back as an HTTP response. The second execution trace of the multivariant binary shows the same sequence of dispatcher and function calls as the previous trace, and also shows that for a different requests, the variants of $f1$ and $f3$ are different.

The key insights from these figures are as follows. First, from a client's point of view, a request to the original or to a multivariant endpoint, is completely transparent. Clients send the same data, receive the same result, through the same protocol, in both cases. Second, this figure shows that, at runtime, the execution paths for the same endpoint are different from one execution to another, and that this randomization process results from multiple random choices among function variants, made through the execution of the endpoint.

C VARIANTS PRESERVATION

During our experiments, we checked for code diversity preservation after compilation. In this work, diversity is introduced through transformation on WebAssembly code, which is then compiled by the Lucet compiler. Compilation might perform some normalization and optimization passes when translating from WebAssembly to machine code. Thus, some variants synthesized by MEWE might not be preserved, i.e., Lucet could generate the same machine code for two WebAssembly variants. To assess this potential effect, we compare the level of code diversity among the WebAssembly variants and among the machine code variants produced by Lucet. This experiment reveals that the translation to machine code preserves a high ratio of function variants, i.e., approx 96% of the generated variants are preserved. This result also indicates that the machine code variants preserve the potential for large numbers of possible execution paths.

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

https://doi.org/10.1145/3358504.3361228